AUTOMATING THE CRACKING OF SIMPLE CIPHERS

by

Matthew C. Berntsen

A Thesis

Presented to the Faculty of
Bucknell University
In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science with Honors in Computer Science
April 19, 2005

Approved: _____

Richard J. Zaccone
Thesis Advisor

_____

Gary Haggard
Chair, Department of Computer Science

Table of Contents

List of Tables

List of Figures

Abstract

There is little point to enciphering information if one does not wish to keep it secret. If one wishes to keep information secret it follows that others would likely want access to that information. It is easy to come up with many scenarios where one would like secret information to be exchanged publicly. For example, Internet shoppers want their credit card information to be secured. Would-be thieves want access to that information. Ciphers are one method of concealing information. Cryptanalysis can identify vulnerabilities in a cipher that help both to break existing ciphers and to create stronger ciphers that will not exhibit the same vulnerabilities.

This project is an examination of methods used to crack the Vigenère cipher with a periodic key, and how these methods can be optimized. Its goals were to make the process as automated and efficient as possible, to explore different methodologies for the various tasks used and to analyze their runtimes both theoretically and experimentally.

It was found that the Vigenère cipher can be consistently broken in $O(N)$ time. Two methodologies were examined; one used only the Friedman test, and the second used both the Friedman and Kasiski tests. The former consistently determined the correct keyword in $O(N)$ time while the latter ran in $O(N^2)$ time and did not consistently find the correct keyword, although it did find it a considerable majority of the time.

Although it is not necessary to use in cracking the Vigenère cipher, the Kasiski test is of particular interest because of its parallels in Computational Biology and data mining. The Kasiski test is also intriguing from an algorithm design and optimization standpoint as it is a difficult algorithm to implement efficiently. This thesis has developed a dynamic programming algorithm to perform the Ka-

siski test that outperforms the brute force alternative in both runtime and memory utilization.

Chapter 1

Introduction

Ciphers have been used throughout history to keep information secret. Text ciphers, such as the Vigenère cipher (Explained in depth in Section 2.4.), work by applying some methodology and a key to transform an English plaintext into a less meaningful ciphertext. Simple text ciphers fall into one of two categories: substitution ciphers, in which each letter of plaintext is replaced by a letter of ciphertext, and transposition ciphers which seek to break up the relationships between neighboring letters by reordering blocks of letters. Substitution ciphers can be either monoalphabetic, where there is a one-to-one mapping of plaintext to ciphertext letters, or polyalphabetic, in which one plaintext letter may be represented by multiple ciphertext letters throughout the ciphertext.

Definitions:

Cipher A method by which plaintext is reversibly transformed into a less intelligible ciphertext.

Ciphertext The text that results from encipherment of a plaintext.

Plaintext A text in ordinary English to be enciphered.

Text Cipher A method of concealing information that works directly on the letters (text) as opposed to their data representation.

After World War II and the advent of the computer, simple text ciphers began to fall out of popular use [3]. With computers came two things: First, the ability to process immensely

more information than a human in a given amount of time, greatly improving the capabilities of brute-force and other attacks and second, data encryptions that act on the bit-wise representations of data, rather than the letters themselves. Data encryptions are advantageous both because they allow the same encryption algorithm to be used on any unit of data as opposed to only text, and because they are more difficult to crack.

The crucial point about ciphers is that one must, given the proper key, be able to decipher the ciphertexts that they produce. As such, they cannot be random, and hence must conform to some algorithmic methodology. This inevitably results in some form of weakness. Monoalphabetic substitution ciphers are vulnerable to frequency analysis and comparatively simple to crack. Polyalphabetic substitution ciphers such as the Vigenère cipher are more difficult to crack.

Definitions:

Frequency Analysis A tool in cryptanalysis that is used in cracking monoalphabetic ciphers. The known frequencies of letters in the English language (Found in Table 2.1.) are compared with the frequencies of the letters in the ciphertext. If a shift can be found where the two sets of frequencies match closely, the cipher used is likely to be an additive cipher with the indicated shift. If not, other tools must be used to crack the cipher.

Key A value used by a cipher to encipher plaintext to ciphertext and decipher ciphertext to plaintext. Although these can take on many forms, keys in this project will be strings of contiguous letters such as "`ABC`" or "`BUCKNELL`".

1.1 Weaknesses of Vigenère

The strength of the Vigenère cipher is almost entirely dependent on the key and its relation to the input text. A key of length one is an additive or shift cipher, which is trivial to break using frequency analysis. Keys that are longer but short compared to the size of the input text must be repeated, which is the case that this project explores. If the key is very long compared to the input text (and non-repeating) or some variant of an autokey cipher, files encrypted with

Vigenère would be virtually impossible to crack as the repeating key produces patterns that are critical in breaking the cipher.

Definitions:

Autokey Cipher A cipher that incorporates the plaintext into the key. This uses a key that consists of a short priming key concatenated with the plaintext.

Shift Cipher Formally known as an Additive Cipher, shift ciphers work by mapping each plaintext letter to a ciphertext letter a fixed number of positions beyond it in the alphabet. For more information, please see Section 2.1.

Given a relatively short repeating key, the weaknesses of the Vigenère cipher are already known and well documented [3,6,7]. The process of cracking the Vigenère cipher traditionally involves a human. This project has built a framework that removes the human aspect from the process of cracking Vigenère and potentially other ciphers through the use of a modularly constructed computer program.

Vigenère can be broken with the use of two tests, the Kasiski test (Section 2.4) and the Friedman test (Section 2.3), which for the sake of this project will assume that texts being examined are written in the English language. The Friedman test consists of two equations which indicate the type of cipher used (monoalphabetic vs. polyalphabetic) and estimate the key length assuming a polyalphabetic cipher with a periodic key. The Kasiski test is premised on the assumption that repeated substring in the ciphertext, particularly of longer length, will likely occur at some multiple of the key length. Because the Friedman test is a set of well defined mathematical equations, there is only one methodology for producing its results. There are, however, multiple ways to approach obtaining the results for the Kasiski test.

Both the Kasiski and Friedman tests estimate the key length. Their results can be combined to deduce the most likely key length. Once one has a hypothesis for the key length L, the ciphertext can be arranged into L columns, with one column per letter in the key. Each column will be a simple shift or additive cipher, for which one can use frequency analysis to determine

the key letter used. These key letters are combined to obtain the key used for encipherment. Once one knows the key, the ciphertext can simply be deciphered. (Please see Section 2.1 for discussion of the shift cipher, and how it can be broken.)

1.2 Applications

Algorithms such as the Kasiski test are widely applicable to the field of Computational Biology, which is the development and application of analysis, modeling and simulation to the study of biological systems. A large part of this field is rapid sequence analysis of DNA and proteins, often with a focus on portions of DNA shared between multiple samples. This type of problem and the one solved by the Kasiski test both involve searching for patterns that are not defined before the search process, and must be determined by the search procedure itself. In some senses, however, problems involving DNA are simpler than that of the Kasiski test as DNA uses a four-character alphabet, and the desired sequences of DNA often have start or end markers [4, 5].

Other applications of a sophisticated string searching and comparison algorithm include data mining, which is the practice of searching large amounts of information for previously unknown and potentially important patterns. This information might be used to adjust market prices of various goods, or predict what kinds of people are likely to want a certain product. It also has applications in the sciences finding patterns that commonly appear with a certain physical phenomena such as a gene that correlates to an increased risk of heart disease. The ways in which an algorithm that finds and extracts patterns for analysis could be applied are virtually endless.

Chapter 2

Algorithms Used

The problem of cracking the Vigenère cipher can be divided into two sub-problems: determining the key length and then determining the key. Also, a third step can be added to verify that the cracked ciphertext is written in the English language.

Once the key length has been estimated using the Friedman test, a range of likely passphrase or key lengths is defined around the estimate. The Friedman test can be used on each length in the range to find the one whose Index of Coincidence or *IC* (discussed in Section 2.3) is closest to that of American English, or the Kasiski test can be used to determine a likely key length.

Once the key length L is known, the ciphertext can be divided into L groups, each containing the letters that were enciphered with a given letter from the key. Each of these groups will correspond to text enciphered by an additive cipher, and can be cracked using frequency analysis to see what the corresponding passphrase-letter is for each group. These letters can then be combined to determine the passphrase used for encipherment.

To check this passphrase, the easiest thing to do is check if the new deciphered text contains whitespace. If it does, those spaces define words, which can simply be looked up in a dictionary.

2.1 Additive or Shift Ciphers

Shift ciphers, formally known as Additive Ciphers, are monoalphabetic ciphers that assign a value K to each plaintext letter (A = 0, B = 1, etc.) and take a shift S as a key. They then assign each plaintext letter K to ciphertext letter J using the formula: $J = ((K + S) \bmod 26)$.

Definition:

Mod or Modulo A mod B is defined as the remainder after A is evenly divided by B.

Sum Square Error Method A method of computing a the goodness of fit of an additive cipher. Given the letter frequencies F_1 to F_{26} found in Section 2.1.1 the frequencies G_1 to G_{26} are calculated for each of the possible 26 shifts. The error E is then calculated as follows for each possible shift. The shift with the lowest error is the best fit.

$$E = \sum_{i=1}^{26} (F_i - G_i)^2$$

Shift ciphers are, however, very easy to crack using frequency analysis. Given a large enough plaintext in ordinary English, the letter frequencies will be similar to the accepted values. As shift ciphers do not jumble the mapping of letters, the relative frequencies of the letters can tell one what shift was used. In this project the Sum Square Error method is used to find the best fit.

2.1.1 Letter Frequencies for English

In English, as any western language, certain letters are used more commonly than others. For instance `e` is used more often than `z`. By analyzing a large amount of text, one finds that these letters each appear with relatively constant frequency. Once one knows these frequencies, they can be used to help break monoalphabetic ciphers.

The letter frequencies below, shown both in a table and a graph, were obtained through analysis of approximately 100MB of text downloaded from Project Gutenberg [13], and are consistent with generally accepted values [3].

Table 2.1: Letter Frequencies

Letter	Frequency
A	0.082
B	0.015
C	0.028
D	0.043
E	0.127
F	0.022
G	0.020
H	0.061
I	0.070
J	0.002
K	0.008
L	0.040
M	0.024
N	0.067
O	0.075
P	0.019
Q	0.001
R	0.060
S	0.063
T	0.091
U	0.028
V	0.010
W	0.023
X	0.001
Y	0.020
Z	0.001

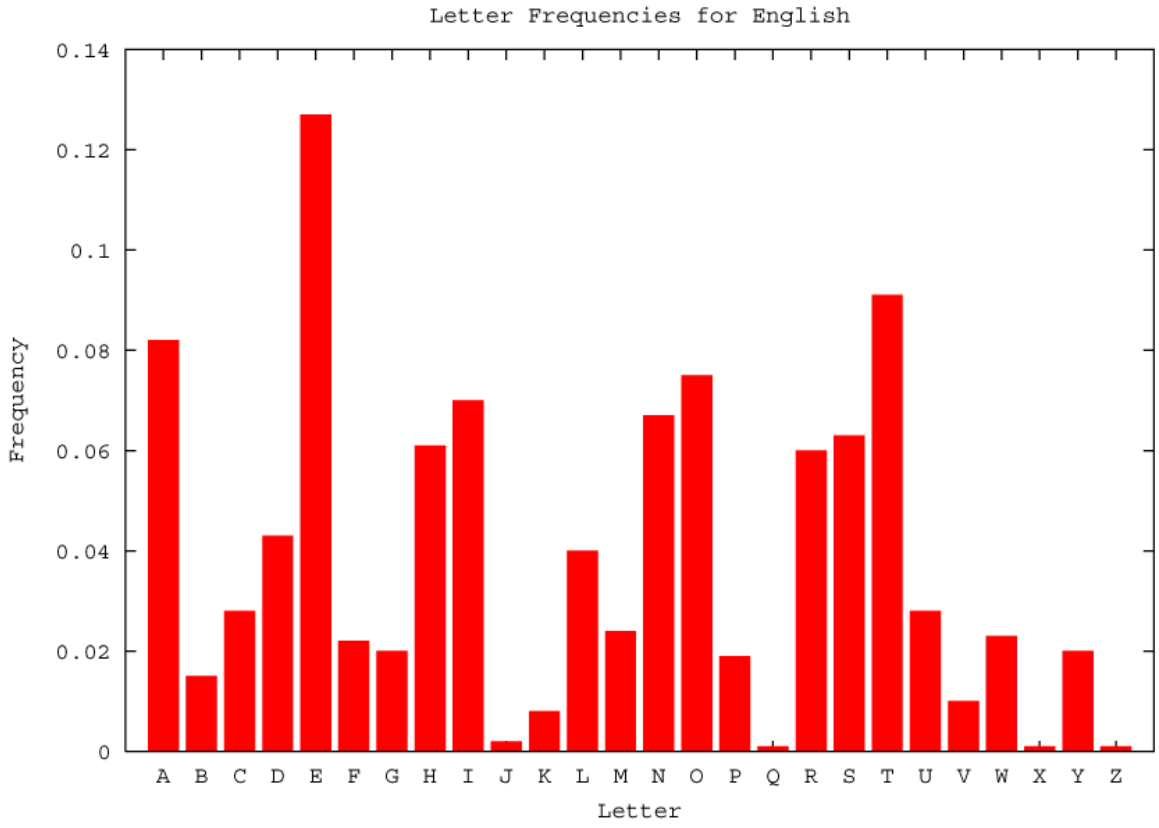
2.2 The Vigenère Cipher

The Vigenère cipher is a simple polyalphabetic cipher based on the tableau in Figure 2.2 below, in which each row of the tableau is shifted one letter to the left from the row above.

2.2.1 Encipherment and Decipherment

The cipher works by taking a keyword, in this case “BUCKNELL”, and uses it to encode a plaintext, “HAS A NICE CAMPUS”. First, the keyword is repeated on top of the plaintext:

Figure 2.1: Letter Frequencies for English



Keyword: BUC K NELL BUCKNE
Plaintext: HAS A NICE CAMPUS
Ciphertext: IUU K AMNP DUOZHW

For each letter in the plaintext, the corresponding ciphertext letter is found by selecting the letter in the column determined by the keyword-letter and the row determined by the plaintext in Figure 2.2.

There are two methods to decrypt the Vigenère cipher. The first simply does the encoding process backwards: Repeat the keyword above the ciphertext, and select the ciphertext letter out of the column determined by the keyword. The row that the ciphertext letter appears in corresponds to the plaintext letter that was enciphered. This is rather time consuming as one

Figure 2.2: The Vigenère Tableau

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

must search the column for the appropriate value.

The second method is to 'encode' again using the inverse of the keyword, which is easier for a computer to do. This is done by taking each keyword letter's numerical value ($A = 0, B = 1$, etc.), subtracting it from 26, and then moding by 26. Once the key is modified as discussed above, the ciphertext is deciphered by running the encipherment algorithm with the inverted key.

2.3 Friedman Test

The Friedman test relies on the concept of the Index of Coincidence (*IC*). Simply put, the *IC* is the probability that when two letters are selected at random, they are the same letter. Given the number of ciphertext letters n, where n_1 = number of A's in the ciphertext, n_2 = number of B's in the ciphertext, etc., the *IC* is defined as follows:

$$IC = \sum_{i=1}^{26} \frac{n_i(n_i-1)}{n(n-1)} \tag{2.1}$$

From the accepted letter frequencies, one finds that the *IC* for the English language is about 0.065. It follows that a monoalphabetic cipher will have an *IC* that is approximately equal to 0.065. If it is not, then the cipher is most likely polyalphabetic [2].

The Friedman test can also be used to estimate the key length of a Vigenère ciphertext. If the ciphertext is placed in columns, with one column for each letter of the keyword, each column can be thought of as a shift cipher of the plaintext equivalent (See Section 2.1). It follows that each column will have the same *IC* as ordinary English. Hence, the key length can be written as a function of the *IC*.

$$keylength \approx \frac{0.027n}{(n-1)IC - 0.038n + 0.065} \tag{2.2}$$

The Friedman test can also be used to find the 'best' key length in a specific range. If one were to divide the ciphertext up into L columns or groups as described above, and then average the *IC*s for the groups, the L with the *IC* closest to that of American English (0.065) should have the best fit, as each group should have an *IC* approximately equal to English if the key length is correct. Hence in finding the best-fit number of groups, we find the best-fit key length.

2.4 The Kasiski Test

In 1863 Major F. W. Kasiski, a German cryptanalyst, published a paper [8] in which he stated that for all periodic ciphers, given a ciphertext of sufficient length, there will be a plaintext letter pattern that is enciphered with the same key pattern such that the resulting ciphertext from each group is the same [6].

2.4.1 How the Kasiski Test Works

To illustrate how the Kasiski test works, consider the plaintext from Shakespeare's 40^{th} sonnet below. The plaintext is enciphered using the Vigenère cipher with the key `CRYPTOGRAPHY` of length 12, producing the ciphertext. Repeated substrings in the ciphertext are underlined.

```
     Key:  CRYP TOG RA PHYCR YP TOGR APH YCRY PTOG RAP HYCR YPTO
Plaintext:  TAKE ALL MY LOVES MY LOVE YEA TAKE THEM ALL WHAT HAST
Ciphertext: VRIT TZR DY AVTGJ KN ECBV YTH RCBC IASS RLA DFCK FPLH

     Key:  GRAP HYCR YPTO GRAP HYCR YPTOG RAPHYC RY PTOG RA PHYC
Plaintext  THOU THEN MORE THAN THOU HADST BEFORE NO LOVE MY LOVE
Ciphertext: ZYOJ AFGE KDKS ZYAC AFQL FPWGZ SEUVPG EM AHJK DY AVTG

     Key:  RYPT OGRA PHYCR YPTO GRAP
Plaintext:  THAT THOU MAYST TRUE LOVE
Ciphertext: KFPM HNFU BHWUK RGNS RFVT
```

For the sake of demonstration, analysis will be limited to repeated substrings of length three. All of the three-letter patterns have been extracted below with their locations. (Spaces are not counted.) The differences between the locations were calculated, and their prime factorizations have been determined.

Substring	Positions	Difference and Factors
DYA	8, 80	$72 = 2^3 \times 3^2$
YAV	9, 81	$72 = 2^3 \times 3^2$
AVT	10, 82	$72 = 2^3 \times 3^2$
VTG	11, 83	$72 = 2^3 \times 3^2$
KFP	38, 86	$48 = 2^4 \times 3^1$

As can be readily determined by the prime factorizations, the largest common factor of the distances between these substrings is $2^3 \times 3^1 = 24$. This is twice the key length of twelve, so to form a best-fit key of length 24, the key would simply be repeated once, forming the key `CRYPTOGRAPHYCRYTOGRAPHY`.

2.5 Determining the Key Length and Key

Although it has been shown how the Friedman and Kasiski tests can each be used to estimate the key length, it is not immediately clear how the information generated by the two tests can be combined. In the course of this project, the processing on the locations done at the end of the previous section is done while considering the result of the Friedman test.

First the floating point length estimate from the Friedman test is converted to an integer, and rounded accordingly. Next a range of potential key lengths is formed around it, spanning 25 values in either direction unless restricted by 0 or the maximum key length value, which is set to 200 by default. This results in a window of (at most) 51 values to be considered as possible key lengths.

Next the differences between the locations are calculated as in the previous section, and their prime factors are weighted so that factors occurring in longer repeated substrings or more often are given precedence.

The prime factors are then sorted by weight, and combined to form the most likely combination within the interval in question. Factors weighted approximately twice that of the next one are multiplied by 2 in the process, etc. Once a final value is settled upon, it is multiplied by the largest natural number that will allow it to remain in the window of lengths being considered. For instance, if the window was 31-82 and the estimate is 35, it would be multiplied by $82/35 = 2$ (the remainder is ignored). This way an additional factor of two is included in the estimate, reducing the possibility of error as a multiple of the key length will still yield the

correct key.

Taking the estimate E the ciphertext is divided into E groups, and frequency analysis is performed on each group to determine the key letter used to encipher the group. These key letters are put together to form a key of length E. This can be easily examined for repeats, so proposed keys such as `CRYPTOCRYPTO` are reduced to `CRYPTO`. The ciphertext is then decrypted with this proposed key, and dictionary verification is run on the result as described in the next section.

2.6 Dictionary Verification

If whitespace has been used in the encipherment to distinguish between words (to allow the intended recipient to have an easier time reading it), once a potential passphrase is generated, the resultant deciphered words can be looked up to see if they are indeed in the desired language, in this case English.

Definition:

Whitespace Any character or series of characters that are displayed as empty space. This includes spaces, tabs and line termination characters.

2.7 Chapter Summary

This chapter has shown how a message enciphered via the Vigenère cipher with a periodic key can be broken. Using the Friedman and Kasiski test, a best-fit key length is determined. If this length L is a multiple of the actual key length, then every L letters in the ciphertext were enciphered with the same key-letter. Any group enciphered with only one key letter has effectively been enciphered with an additive cipher, which can be easily broken with frequency analysis. Once all of the key-letters are obtained, they can be put together to form the key. The

file can be decrypted with this key and its contents can be checked against a dictionary to verify that it is written in English and that the guessed key is correct.

Chapter 3

A Dynamic Programming Algorithm for Performing the Kasiski Test

The fundamental problem that was addressed in this thesis was how to implement an algorithm for the Kasiski test that is efficient both in the time that it takes to execute on a given file and its consumption of memory. A brute force or naive algorithm was used as a base of comparison for a dynamic programming algorithm that was developed to utilize the optimal substructure and overlapping subproblems inherent in the Kasiski test. Through the use of memoization, previous iterations are used to construct the current iteration more efficiently.

This chapter works through both algorithms, outlining specifically how they work. An examination into the memory utilization of each algorithm is also conducted. Theoretical and experimental runtime analyses can be found in Chapters 4 and 5 respectively. There is no discussion of the data structures used as they are essentially the same for both algorithms: A list of strings that corresponds to a list of locations where they can be found.

The algorithms below describe the implementations that can be found in the code (see Section C). **Control sequences** are in bold, *variable names* are in italics and other instructions are in normal text. Single equals (=) indicates assignment and double equals (==) indicates

an equivalence test. Comment lines begin with the # character. Discussion of the processing done on stored lists of repeated substrings is not provided as that aspect of the algorithms is identical. In the examples, the positions start at 1.

Definitions:

Array A method of storing a known number of variables contiguously in memory such that each variable contained inside can be accessed by name in $O(1)$ time.

Binary Search A method of searching a sorted array that halves the area of the array being examined in every iteration. Searches run in $O(\log N)$ time.

Linked List A dynamic method of storing an unknown number of variables in memory. Only the start point and the next variable in the linked list are known at by any node, so accessing any specific variable inside will take $O(N)$ time.

Runtime A way of describing how the time necessary for a program or function to run changes in relation to the size of the input. The Big-O Notation describes an asymptotical upper bound for the runtime as N approaches infinity. Formally, a function is $O(g(N))$ if and only if for its runtime F there exist positive constants C and N_0 such that $0 \leq F \leq Cg(N)$ for all $N \geq N_0$. For the duration of this thesis N will denote the size of the input to a given function or algorithm, which is often the number of characters in a file being analyzed.

3.1 Brute Force Algorithm

This algorithm completes the Kasiski test naively, without taking advantage of the nature of the problem being solved. It is possible to use work done in previous iterations to reduce the amount of work done in future iterations. This allows the implementation to be comparatively simple, but causes it to be less efficient than the dynamic programming alternative both in terms of runtime and memory usage.

3.1.1 Algorithmic Notation

1. **read** input data into *input string*
2. *size* = 4
3. **while** *repeated substring list* is not empty or *size* == 4 **do**

```
4.    empty repeated substring list
5.    for every successive substring S in input string of length size do
6.        found = false
          # Check to see if the substring has already been found to repeat.
7.        for every substring T in repeated substring list do
8.            if S == T then
9.                found = true
          # If not a known repeat, check remaining text for repeats.
10.       if found == false then
11.           for every subsequent substring R in input string do
12.               if S == R then
13.                   add S and R along with their locations to repeated substring list
14.   store repeated substring list for processing
15.   size = size + 1
```

This algorithm works by starting with substrings of length $L = 4$ and searching until a length is found with no substrings. In each of these iterations each substring of length L is considered. First *repeated substring list* is searched. If the substring is found, it is ignored. If the substring is not found, the remainder of the input is searched for other occurrences of the substring, and if any are found they are entered along with their locations into *repeated substring list*. This process is then repeated for all subsequent substrings of length L.

This algorithm is extremely simple and straightforward, however the current iteration does not make use of the work already done, resulting in unnecessary inefficiencies. As it potentially stores the entirety of the input file twice in memory, once in *input string* and again in the *repeated substring list*, as well as the locations of each occurrence, it seems that its memory usage could be improved upon. Also, as it has to potentially compare every single substring of length *size* on every iteration, it is finding the same information multiple times, greatly increasing the runtime.

3.1.2 Sample Execution

A very short and extremely simplified plaintext has been used because of the complexity of a longer example would make this difficult to understand. As future iterations do not build on previous ones, only the iteration for *size* = 4 has been shown. The maximum length of the repeated substrings is five for this example, although an iteration is potentially run for every substring length greater than or equal to four, which is bounded by the size of the input file.

Key:	CRYPT CRYPT CRYPTCRYP
Plaintext:	ROMEO ROMEO WHEREFORE
Ciphertext:	TFKTH TFKTH YYCGXHFPT

The contents of *repeated substrings list* and are shown below for all of the $N-4$ iterations involved in moving through the ciphertext. The substrings are shown with positions in parenthesis, separated by comas.

Position:	(1) TFKTH TFKTH YYCGXHFPT
Repeated:	*empty*
Position:	(2) TFKTH TFKTH YYCGXHFPT
Repeated:	*empty*
Position:	(3) TFKTH TFKTH YYCGXHFPT
Repeated:	*empty*
Position:	(4) TFKTH TFKTH YYCGXHFPT
Repeated:	*empty*
Position:	(5) TFKTH TFKTH YYCGXHFPT
Repeated:	*empty*
Position:	(6) TFKTH TFKTH YYCGXHFPT
Repeated:	TFKT (1, 6)
Position:	(7) TFKTH TFKTH YYCGXHFPT
Repeated:	TFKT (1, 6), FKTH (2, 7)
Position:	(8) TFKTH TFKTH YYCGXHFPT
Repeated:	TFKT (1, 6), FKTH (2, 7)

As there are no more repeated substrings between this point and the end, we will skip to after the final iteration.

Position: (16) TFKTTH TFKTH YYCGXHFPT
Repeated: TFKT (1, 6), FKTH (2, 7)

This process searches the remaining text for all but two of the sixteen iterations. The process of searching through the repeated substrings and the entirety of the file for every insertion can be very time consuming, especially as the process must be repeated for additional key lengths of five, six, etc. Fortunately, there is a way to leverage work already completed to greatly increase the speed of subsequent iterations.

3.2 Dynamic Programming Algorithm

The problem of the Kasiski test is finding all of the repeated substrings in a file in a range of lengths. The task of eliminating the inefficiencies of the brute-force algorithm will be comparatively very complex. This dynamic programming algorithm utilizes the optimal substructure and overlapping subproblems inherent in the Kasiski test to optimize both its runtime and memory usage.

The nature of the overlapping subproblems in the Kasiski test is relatively straightforward - any pattern of letters with a length $L \geq 4$ is composed of multiple groups of letters. In this case, we consider a pattern of length $L-1$ and a pattern of length 3, with the pattern of length 3 sharing the last two letters of the other pattern. If a single substring can be built from two smaller substrings, repeated substrings of length L can be found by considering repeated substrings of length $L-1$ and length 3.

The optimal substructure in the Kasiski test is based first on the nature of the overlapping subproblems, and the fact that the problem in question is not finding only the longest repeated substring, but a range thereof. This problem is compounded when the nature of the range is not known prior to runtime, as it typically is with the Kasiski test, although the lengths being examined could be fixed. Because the test requires numerous iterations, and does not know

what the final iteration will be, it is necessary to compute the iterations starting at a relatively small substring size. As the number of unique (ie. `aaa, aab,` etc.) repeated substrings will generally tend to decrease as the size being considered grows. This is due to the structure of the English language, where words like `the` are more likely to be repeated than specific 100 or more character combinations. Ultimately, the requirement of optimal substructure is satisfied as the previous iterations are necessary, and can be easily leveraged to computer the current iteration more efficiently.

As the algorithm utilizes the optimal substructure and overlapping subproblems involved in the Kasiski test, it satisfies the requirements of a dynamic programming algorithm.

3.2.1 Algorithmic Notation

The initial processing functionality of this algorithm makes use of the fact that there are $26^3 = 17,576$ possible three letter combinations. Because of this, an array can be used to allow each occurrence of a 3-letter pattern and its position to be read into memory in lexicographical order in $O(1)$ time. Also, the location in the array serves as a placeholder for the actual substring, so only the locations need to be stored in memory. Because it is already sorted and the number of elements E is bounded by a constant, this allows for searches via binary search after the non-repeated substrings are removed in $O(\log E) = O(1)$ time.

1. *max key length* = 200
   # Build list of all 3-letter substrings and their locations.
2. **read** input file
   # Remove all 3-letter non-repeated substrings.
3. **for** every successive 3-character substring in *input file* **do**
4. insert location into *3-letter position list*
5. *counter* = 0
6. **for** every element in *3-letter position list* **do**
7. **if** there is more than one location in this element **then**

```
        counter = counter + 1
repeated 3-letter array is set to length counter
for every element in 3-letter position list do
    if there is more than one location in this element then
        insert this element's locations into repeated 3-letter array
# Build linked list of 4-letter repeated substrings. Store them in current repeated list.
for every element in repeated 3-letter array do
    for i = 0 to 25 do
        compare the location lists of the element and the one that corresponds to its latter two
        characters plus the character determined by i for separations of 1
        if 2 or more matches were found then
            add the 4-letter pattern and its locations to current repeated list
store current repeated list for processing
size = 5
# Perform iterations for lengths 5, 6, etc. Store them in current repeated list.
while size ≤ max key length and current repeated list is not empty do
    old repeated list = current repeated list
    empty current repeated list
    for every element in old repeated list do
        compare the location lists of the element and the one from repeated 3-letter array that
        corresponds to its latter two characters plus the character determined by i for separations
        of size - 3
        if 2 or more matches were found then
            add the size-letter pattern and its locations to current repeated list
    store current repeated list for processing
    size = size + 1
```

The algorithm starts by generating all repeated substrings of length $L = 3$. As there are only $26^3 = 17,576$ combinations of three letters, this can easily fit in memory, which allows the list of repeated substrings and locations to be built extremely efficiently.

The insight that any 4-letter combination is simply a combination of two 3-letter patterns results in the optimal substructure and overlapping subproblems necessary for a dynamic programming algorithm. Because of this, the list of repeated 4-letter substrings can be built from the array of 3-letter substrings (which is left in an array rather than a linked-list to optimize searching).

Once a list of repeated substrings of length 4 is built, the lists of length 5, 6, etc. are built using the previous iteration and the array of repeated 3-letter substrings. As each iteration likely reduces the total amount of repeats (unique locations rather than combinations), using the previous iteration allows the algorithm to ignore information that is not potentially useful, resulting in better performance.

The decision to begin by storing repeated three-letter patterns and using this information to build subsequent pattern sizes potentially uses more memory than the brute force algorithm at first. This initial inefficiency provides the ability the work without the entirety of the file in memory, allowing searches to only consider known repeated substrings which greatly reduces the runtime as text that is not repeated in a given substring length is omitted from subsequent iterations.

The variable *old repeated list* contains only repeated substrings of length *size* - 1. By using *old repeated list* and *repeated 3-letter array* to build *current repeated list*, the overlapping subproblems are leveraged to make the current iteration more efficient. As non-repeated data of length *size* - 1 is not considered, there is no need to store it in memory, making the dynamic programming algorithm's average memory utilization considerably better than that of the brute force algorithm as the number of iterations increases.

3.2.2 Sample Execution

This will show the execution through completion for the same sample ciphertext as the brute force algorithm.

Key: CRYPT CRYPT CRYPTCRYP
Plaintext: ROMEO ROMEO WHEREFORE
Ciphertext: TFKTH TFKTH YYCGXHFPT

First, the 3-letter combinations are placed with their locations into an array in lexicographical order. Patterns with no locations are omitted from the list of tuples below, each containing a string and list of positions.

3-Letter Position List: CGX (13), FKT (2, 7), FPT (17), GXH (14), HFP (16), HTF (5), HYY (10), KTH (3, 8), TFK (1, 6), THT (4), THY (9), XHF (15), YCG (12), YYC (11)

Next, the repeated patterns are copied to *repeated 3-letter array*.

Repeated 3-Letter Array: FKT (2, 7), KTH (3, 8), TFK (1, 6)

Find 4-letter patterns. For FKT in *repeated 3-letter array*, search for all 3-letter patterns KT* where * is a letter. Looking at the list above, the only one of the form KT* is KTH. If any of the locations for KTH are one greater than a location for FKT, then the 4-letter pattern FKTH has been found. This occurs in the combination of the tuples FKT (2, 7) and KTH (3,8), resulting in the tuple FKTH (2, 7). Repeat this process for all subsequent patterns in *repeated 3-letter array*, storing the 4-letter patterns found in *current repeated list*.

Current Repeated List: FKTH (2, 7), TFKT (1, 6)
Old Repeated List: *empty*

Now *size* is incremented and *current repeated list* is copied to *old repeated list* and then emptied. The process is repeated for all of the (*size*-1)-letter patterns in *old repeated list*, finding all of the 3-letter patterns that match each element's last two letters. This is done using *repeated 3-letter array* because sorted arrays allow for fast searching via binary search. After this process is run for *size* == *5*, the lists are as follows:

Current Repeated List: TFKTH (1, 6)
Old Repeated List: FKTH (2, 7), TFKT (1, 6)

This process of analyzing *old repeated list* to form the new *current repeated list* is repeated for increasing lengths until there are no patterns found of length *size*. In this example, this happens when *size* == 6.

Current Repeated List: *empty*
Old Repeated List: TFKTH (1, 6)

At this point, the same information that the brute force algorithm found has been determined. The statistical likelihood of a ciphertext containing repeated substrings decreases with the size of substrings being examined, so the amount of data considered usually decreases in successive iterations. Because of this, the process is considerably faster and more efficient in terms of memory than its brute force counterpart.

3.3 Chapter Summary

The Kasiski test can be implemented in a number of different ways which largely fall into two categories: brute force and dynamic programming. The brute force or naive method is elegant in its simplicity, but is cumbersome in terms of runtime and memory usage. For this thesis a dynamic programming algorithm was developed that makes use of the optimal substructure and overlapping subproblems inherent in performing the Kasiski test for successive substring lengths. The dynamic programming algorithm is much more efficient both in terms of memory usage and runtime, but its implementation is much more complicated.

Chapter 4

Theoretical Runtime Analysis

When designing an algorithm it is important to strike a balance between the amount of memory used and the time that the algorithm takes to complete. There are many different ways to make algorithms more efficient, but efficiency usually comes at the cost of complexity. As such, any discussion of an algorithm's runtime is integrally linked with the way that it is implemented.

This chapter discusses the algorithms used as a means to theoretically determine their Big-O runtime. This information is verified by the experimental analysis found in Chapter 5.

4.1 Preparation

In the preparation process, characters are read in, and if they are legal they are written to the output file. Multiple whitespace characters in sequence are reduced to a single space, and may be discarded entirely depending on the mode. This results in a runtime of $O(N)$.

4.2 Encipherment and Decipherment

The process of encrypting or decrypting a single letter executes in constant time whether using the tableau or computational method. Therefore, the process of enciphering or deciphering N

characters will execute in $O(N)$ time.

4.3 Crack Multiple Shift Ciphers

The crack multiple shift ciphers algorithm divides the input file into L groups, assuming that each group had been enciphered with an additive cipher. The algorithm performs frequency analysis on each group by tallying the number of each letter occurring in that group, and trying each of the 26 possible shifts to determine which shift best fits that group.

The process of creating the structure in memory to store the letter occurrences executes in $O(L)$ time, while the grouping and tallying process for an input file executes in $O(N)$. The process for determining the best-bit shift for each group executes in $O(1)$ time, as the calculation time is not affected by the number of characters in the input file. This is done to all L groups, so the entire algorithm executes in $O(L+N)$ time. As L is bounded by the maximum key length, set by default to 200, this runtime simplifies to $O(N)$.

4.4 The Friedman Test

Using the Friedman test to calculate the *IC* and the estimated key length necessitates one read through the file, which executes in $O(N)$ time. As it reads through the file the occurrences of the individual letters are tallied, and these sums are used to compute the *IC* and then the estimated key length. Hence, all aspects of the Friedman test run in $O(N)$ time.

4.5 The Kasiski Test

The Kasiski test is implemented by searching the ciphertext for all three-letter patterns. Any four or five-letter pattern must begin with a three-letter pattern, so in searching for longer matching patterns we can limit the search to extrapolate the patterns already found. This greatly

reduces the number of patterns stored simultaneously in memory. The dynamic programming algorithm used in this project runs in $O(N^2)$ time while the brute-force algorithm used for comparison runs in $O(N^3)$. The individual steps and runtimes are discussed below.

4.5.1 Brute Force Algorithm

The brute force algorithm is naive in that it does not use the information from previous iterations to reduce the amount of data to be analyzed in the current iteration. With this naivety comes an almost stunning simplicity paired with a considerably longer runtime than that of the dynamic programming implementation.

The brute force algorithm begins by searching for all patterns of length $L = 4$. It starts with the first four characters, and moves to each subsequent group until it reaches the end of the file. It stores an unsorted list of the text and locations of each group, which is searched every time a new group is considered. If the group being considered is in the list (and is therefore a repeat) its location is added to the corresponding list element. If it is not a repeat, it is added to the end of the list to be compared with later patterns. The process of searching a list bounded by N items for each of $(N-L)$ subsequent patterns searching for repeats executes in $O(N^2)$ time.

This process is repeated for subsequent lengths until there are no repeats found. The length of a repeated pattern is bounded by $(N-1)$ where all letters are the same, this adds another factor of N for an overall runtime bounded by $O(N^3)$.

4.5.2 Dynamic Programming Algorithm

The dynamic programming implementation of the Kasiski test makes use of the optimal substructure, leveraging previous iterations to simplify the task of performing the current one. It executes the following steps, with line numbers mentioned referring to the algorithm laid out in Section 3.2:

1. *Lines 1-4* - Read through the entire file and write to memory the location and contents of every three-letter substring found. This takes $O(N)$ time, where N is the number of characters in the input file.

2. *Lines 5-12* - Read through this data, eliminating any substring (such as `aaa`) that was not repeated. As there are only $26^3 = 17,576$ possible three-letter substrings, this takes $O(1)$ time.

3. *Lines 13-18* - As every four-letter substring is composed of two three-letter substrings which share the center two letters, it is possible to find all repeated four-letter substrings from the repeated three-letter substrings and locations. To find the locations of a specific substring, for instance `aaab`, the locations of `aaa` and `aab` must be compared and all locations where the former appears one position before the latter are occurrences of `aaab`. For each substring of length L being compared, there are at most $(N-L+1)$ locations at which it occurs in the file. This comparison takes $O(N)$ time.

4. *Lines 19-28* - This process is repeated until there are no more repeated substrings or for all lengths L up to the maximum key length value, which is set by default to be 200. The number of unique (i.e. `aaa`, `aab`, etc.) substrings of any length L is bounded by N. The time to perform the comparisons described in the previous step for all 26 possible substrings of length $(L+1)$ which begin with each specific substring of length L is bounded by $O(N^2)$ as there are at most $26 \times N$ comparisons, each bounded by $O(N)$.

5. During this procedure, the locations of the largest three substring lengths are stored in memory. When the process of finding lengths in finished, these three sets of positions are analyzed to find the prime factorization of the distance between each pair of positions, which occurs in $O(N^2)$ time for each of the three sets of locations. These prime factorizations are used to weight a list of prime numbers according to likelihood that the

prime in question is a factor of the key length. These weights, along with the estimated key length from the Friedman test, are used to find the most likely keyword length, or a multiple thereof. This calculation takes $O(1)$ time.

Considering the runtime of each step outlined above, the dynamic programming implementation of the Kasiski test will have a runtime bounded by $O(N^2)$.

4.6 Dictionary Verification

This project makes use of a dictionary containing more than $160,000$ words, amounting to more than 1.5MB of data. Given the amount of data, a process had to be developed to optimize the creation and searching of a structure in memory.

By assuming that all input will be in lexicographical order and that all insertions will be done before searching begins, it is possible to complete all insertions in $O(1)$ time and searches in $O(\log D)$ time where D is the size of the dictionary. The dictionary is built by inserting words at the beginning of a linked list, and binary search is used to search the sorted array. The first search executes a function that moves the data from the linked list to an array for searching in $O(D)$ time and forbids future insertions. If N varies directly with the size of the file being looked up word by word, the total runtime is $O(D + N \log D)$. As the dictionary size D is constant, the process of creation and a search for every word in a file will run in $O(N)$ time.

4.7 Cracking Functionality

The algorithm used to crack the Vigenère cipher makes use of all of the algorithms above, except for the brute force implementation of the Kasiski test. As Big-O notation is a worst-case analysis, the runtime for the dynamic programming Kasiski test must be included in the overall runtime even if it is not used. The Kasiski test is used only once, and its $O(N^2)$ runtime

is greater than those of all other algorithms in question. Thus, the runtime of the Kasiski test will dominate the runtime, so the cracking functionality of this project has a runtime of $O(N^2)$.

4.8 Chapter Summary

Analysis of algorithms is an important part of Computer Science as it allows us to predict how the runtime of an algorithm will change in relation to the size of the input. An analysis of the algorithms used in this project has been conducted, and it was found that preparation, encryption, decryption, cracking of multiple shift ciphers, the Friedman test and dictionary lookup all run in $O(N)$ time. The Kasiski test is bounded by $O(N^2)$. The runtime of the Kasiski test will dominate the cracking functionality of the project, resulting in $O(N^2)$ runtime for cracking a file encrypted with the Vigenère cipher.

Chapter 5

Experimental Runtime Analysis

Any discussion of theoretical runtimes inevitably begs one to ask whether or not they hold in practice. All of the documents that were used for testing were downloaded as ASCII from Project Gutenberg [13] and were in the English language. Although each group of tests (i.e., all of the tests for encryption, decryption, etc.) were run at the same time under the same load conditions and on the same machine, not all test groups were run on the same machine or at the same time. Upper and lower 95% certainty bounds have been computed for all tests, but as they only differ from the means by at most 0.1% only the means are given.

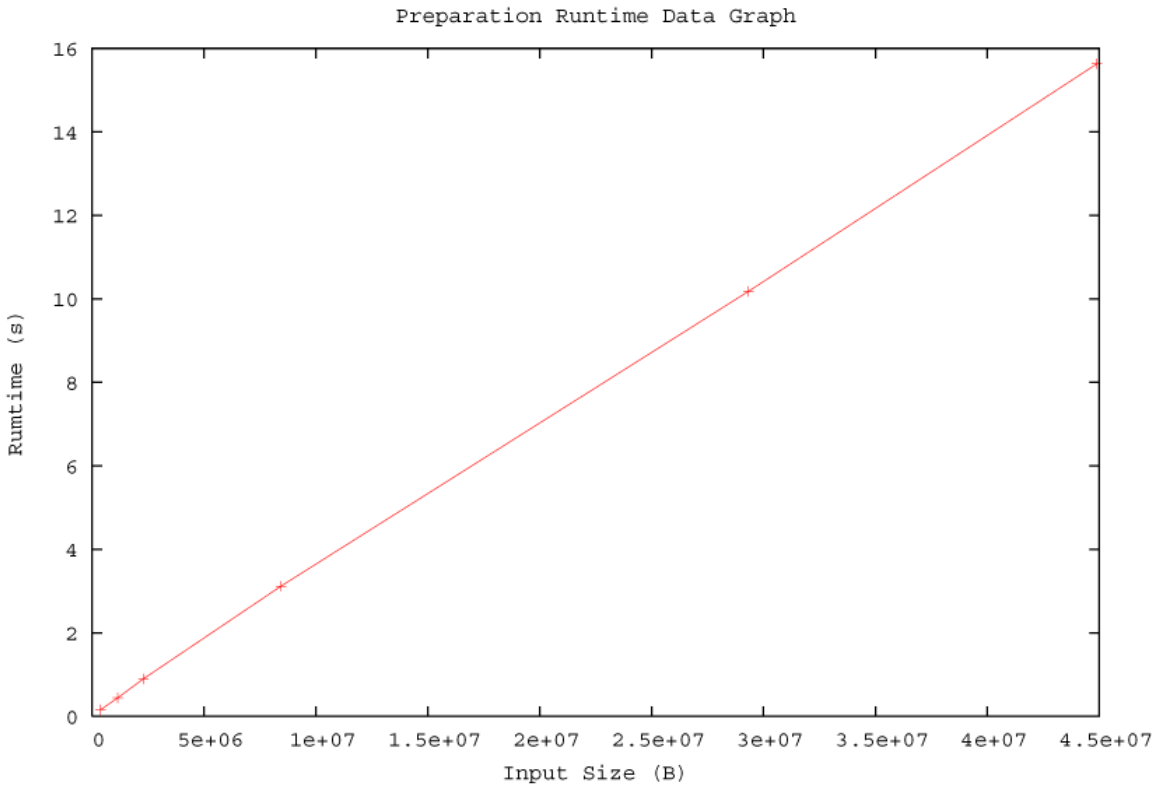
5.1 Preparation

Both modes of input file preparation (include or exclude spaces) run in $O(N)$ time as they do a single pass on the file that they are processing. The time difference for the two modes was negligible, so only one value is included. The file sizes are different than those in other tests because these files have not yet had all non-letters removed. As can be seen in Figure 5.1, this algorithm's experimental runtimes grow linearly as predicted.

Table 5.1: Preparation Runtime Data

Size (B)	Runtime (s)
385,963	0.16
1,146,926	0.44
2,293,852	0.90
8,441,343	3.12
29,325,065	10.18
44,894,190	15.63

Figure 5.1: Preparation Runtime Data Graph



5.2 Encipherment

Both methods of encryption (tableau and computational) theoretically run in $O(N)$ time. The numbers below (1 & 2) denote the mode used, where 1 utilizes the tableau encipherment method and 2 utilizes the computational method.

Definition:

Cache Extremely fast memory used in a computer to store frequently accessed data or instructions.

As the Figure 5.2 below illustrates, both modes grow linearly, with the tableau method being slightly faster. This is likely because the entirety of the tableau can be stored in cache. As the computational method requires multiple processor cycles, it is not as fast as a single cache lookup.

This performance is likely to change if the amount of cache available is smaller than the tableau, however experiments were not conducted as this is a tangential concern. Although all recent computers have more than the requisite amount of cache ($\approx 1KB$), portable devices such as PDAs may not. Performance comparisons between the two algorithms for encryption and decryption with varying cache sizes could be a topic of future research.

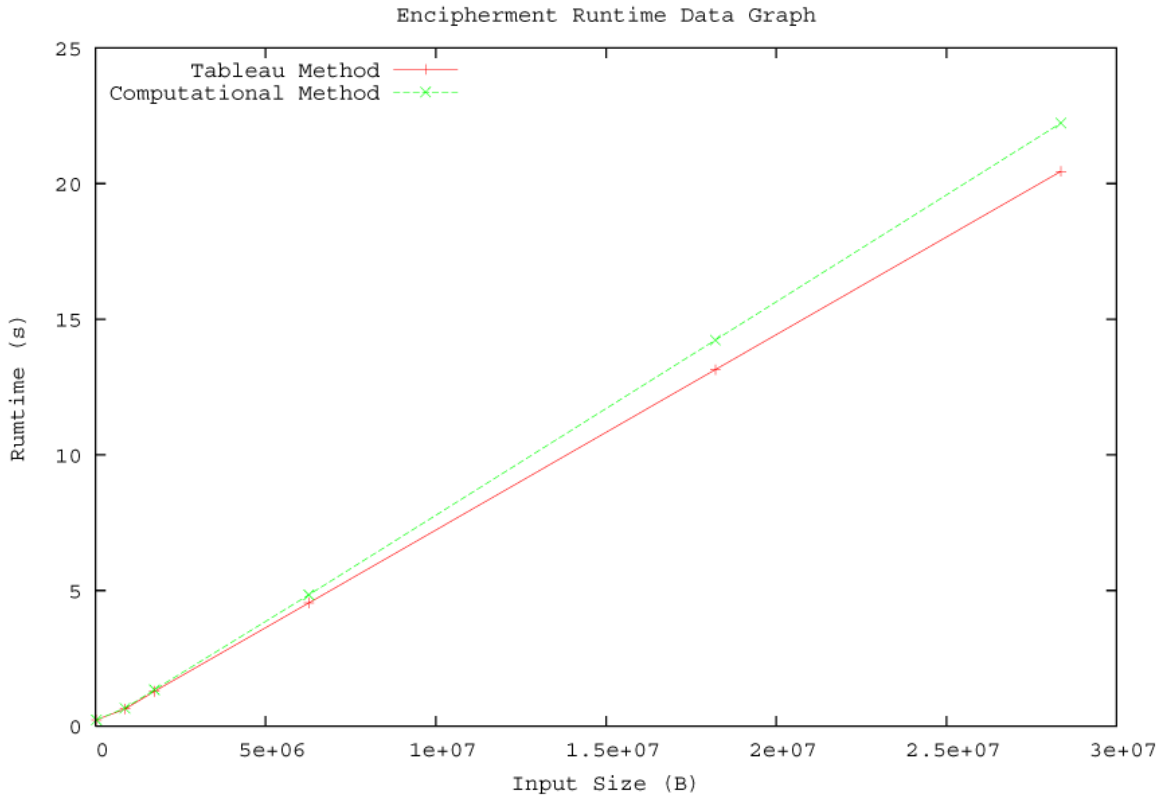
Table 5.2: Encryption Runtime Data

Size (B)	**Runtime 1 (s)**	**Runtime 2 (s)**
286,999	0.23	0.23
865,409	0.63	0.66
1,730,818	1.28	1.34
6,267,581	4.54	4.8
18,210,861	13.14	12.23
28,363,348	20.45	22.23

5.3 Decipherment

Both methods of decryption (tableau and computational) experimentally agree with their theoretical runtime of $O(N)$ as can be seen in Figure 5.3. The reason that mode 1 (tableau lookup) takes considerably longer is that it must search the column for the corresponding ciphertext letter to find the corresponding row, resulting in multiple cache accesses. This process runs in $O(1)$ time as there are only 26 letters, but the process of searching makes the constant consid-

Figure 5.2: Encipherment Runtime Data Graph

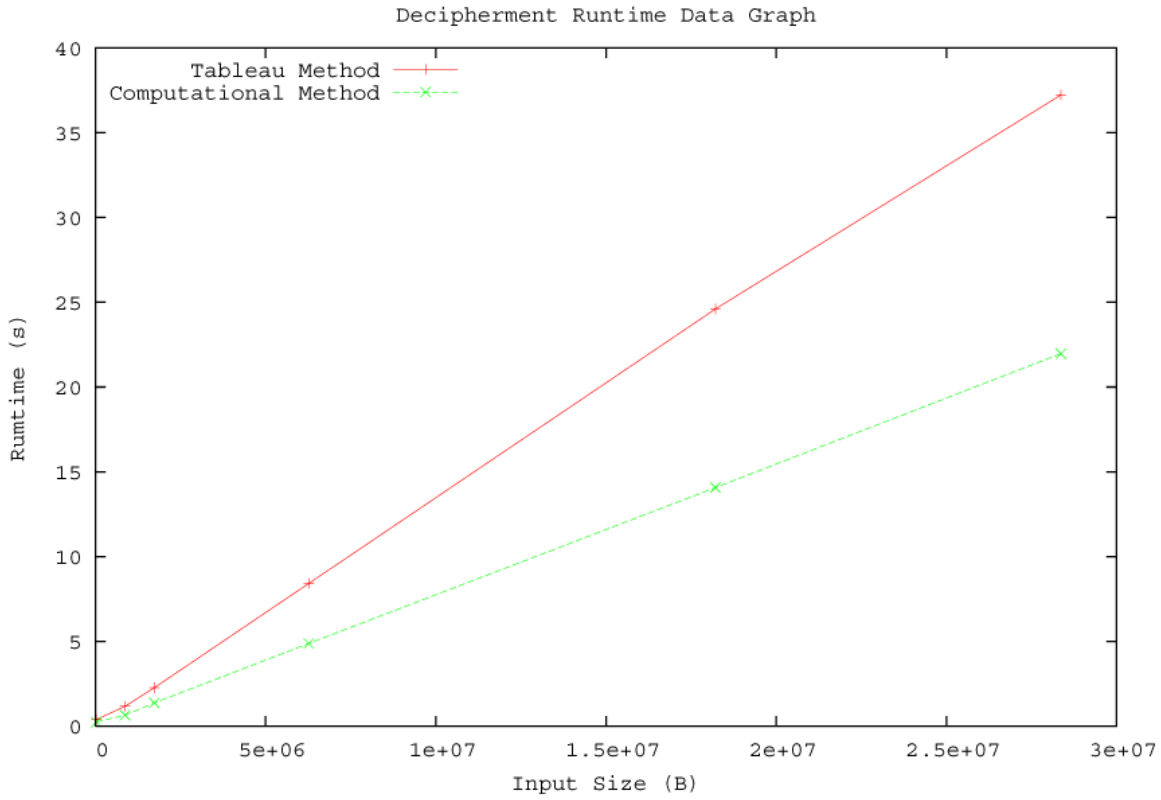


erably larger. The numbers below (1 & 2) denote the mode used, where 1 utilizes the tableau encipherment method and 2 utilizes the computational method.

Table 5.3: Decryption Runtime Data

Size (B)	**Runtime 1 (s)**	**Runtime 2 (s)**
286,999	0.3	0.24
865,409	1.17	0.65
1,730,818	2.27	1.37
6,267,581	8.42	4.88
18,210,861	24.60	14.07
28,363,348	37.23	21.97

Figure 5.3: Decipherment Runtime Data Graph



5.4 Crack Multiple Shift Ciphers

This algorithm divides the ciphertext being cracked into L groups and finding the passphrase letter associated with each group runs in $O(N+L)$ time. Please note that for this project $L \leq 200$, so $L = O(1)$ and the runtime simplifies to $O(N)$. This theoretical runtime is in agreement with the experimental runtimes seen in Figure 5.4.

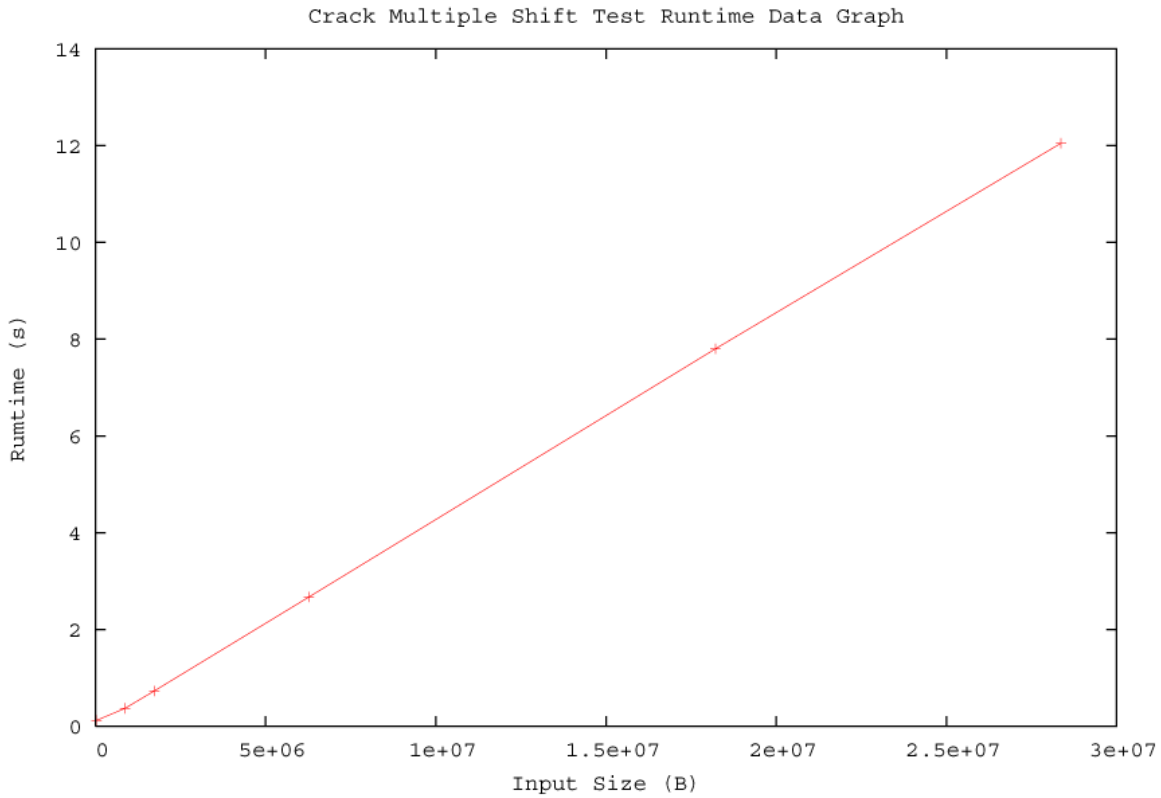
5.5 Friedman Test

The simple mathematical form of the Friedman test theoretically runs in $O(N)$ time. The test timed here only determines the estimated key length, which involves one calculation of the IC. Figure 5.5 indicates agreement with the $O(N)$ theoretical runtime.

Table 5.4: Crack Multiple Shift Runtime Data

Size (B)	Runtime (s)
286,999	0.12
865,409	0.37
1,730,818	0.73
6,267,581	2.67
18,210,861	7.80
28,363,348	12.05

Figure 5.4: Crack Multiple Shift Test Runtime Data Graph



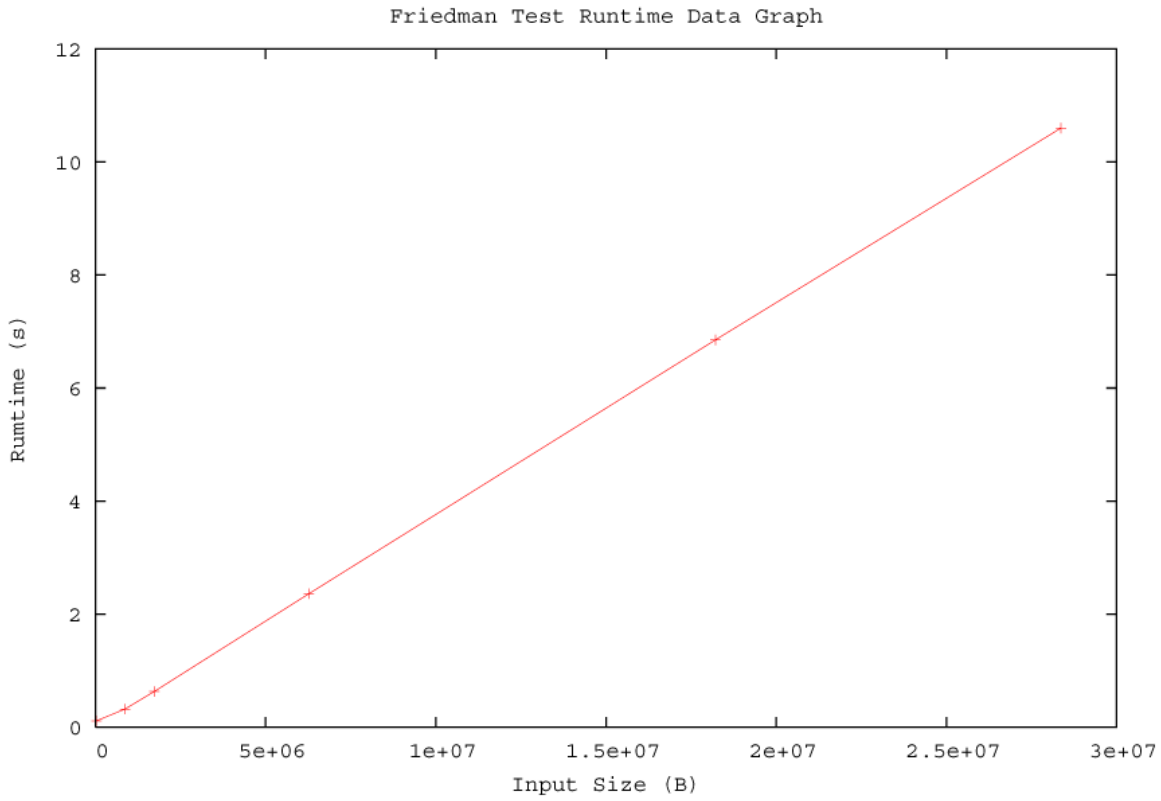
5.6 Kasiski Test

The dynamic programming implementation of the Kasiski test written for this thesis was compared to a brute force implementation to determine the difference in efficiency. As is shown in Figure 5.6, the dynamic programming version was bounded by $O(N^2)$ but behaved close

Table 5.5: Friedman Runtime Data

Size (B)	**Runtime (s)**
286,999	0.11
865,409	0.32
1,730,818	0.63
6,267,581	2.36
18,210,861	6.85
28,363,348	10.60

Figure 5.5: Friedman Test Runtime Data Graph

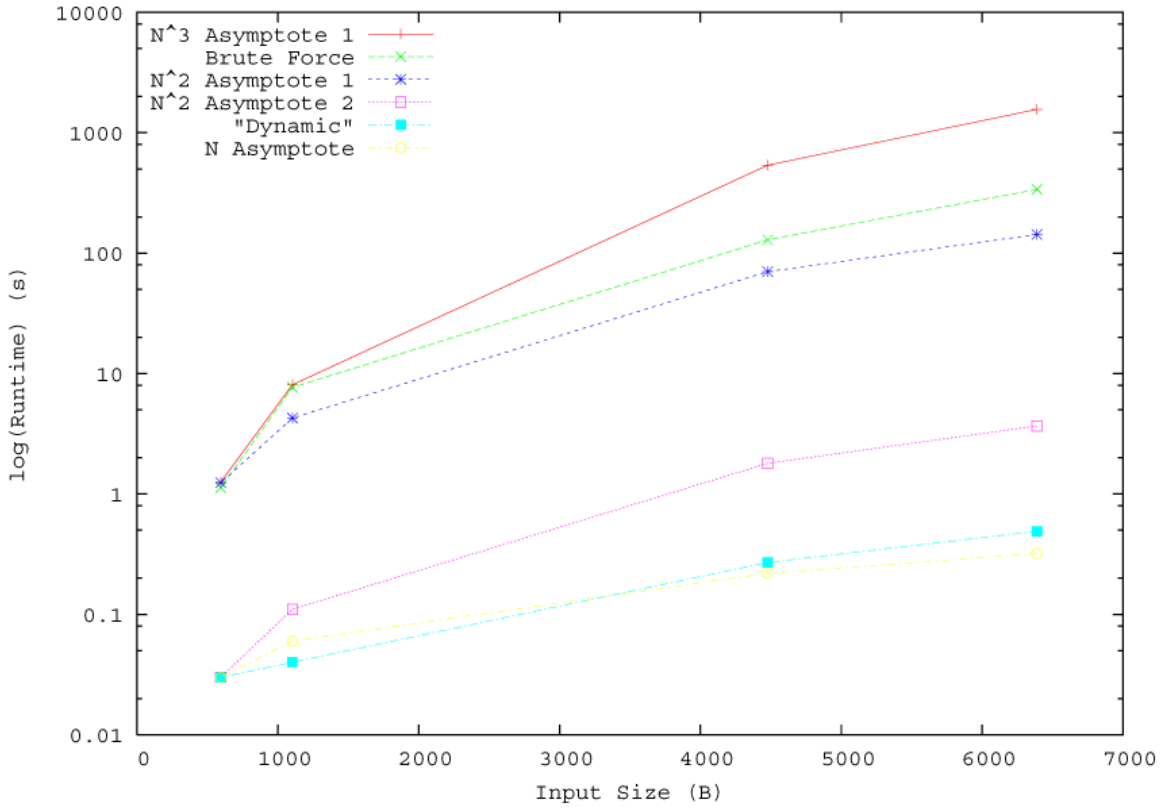


to $O(N)$ while the brute force algorithm was bounded by $O(N^3)$ while behaving more like $O(N^2)$. It should also be noted that the constants involved in the bounds for the dynamic programming version are considerably smaller, which is why its N^2 asymptote appears below the asymptote of the brute force algorithm. The asymptotes on the graph were arbitrarily selected to emphasize the comparison of the algorithms' runtime behaviors.

Table 5.6: Kasiski Runtime Data

Size (B)	Dynamic Runtime (s)	Brute Force Runtime (s)
6,389	0.49	338.25
4,477	0.27	128.96
1,104	0.04	7.68
595	0.03	1.13

Figure 5.6: Kasiski Test Runtime Data Graph



5.7 Dictionary Test

The dictionary included with this project contains 161,838 words in lexicographical order. It initializes in $O(D)$ time, where D is the number of words, which in my tests took ≈ 1.28 seconds. Given that this is building a 2MB array in memory that time is not unreasonable. The first search completes the initialization process by running a function that prepares the array

for searching in $O(D)$ time. All subsequent searches run in $O(\log D)$ time. Thus the overall runtime for dictLookup is $O(D + N \log D)$ where N is the number of words being looked up. If we take D to be a fixed value (161,838) this simplifies to $O(N)$ time.

After adjusting for initialization, Figure 5.7 shows that the dictionary test's runtime grows linearly, validating the $O(N)$ theoretical runtime.

Table 5.7: Dictionary Test Runtime Data

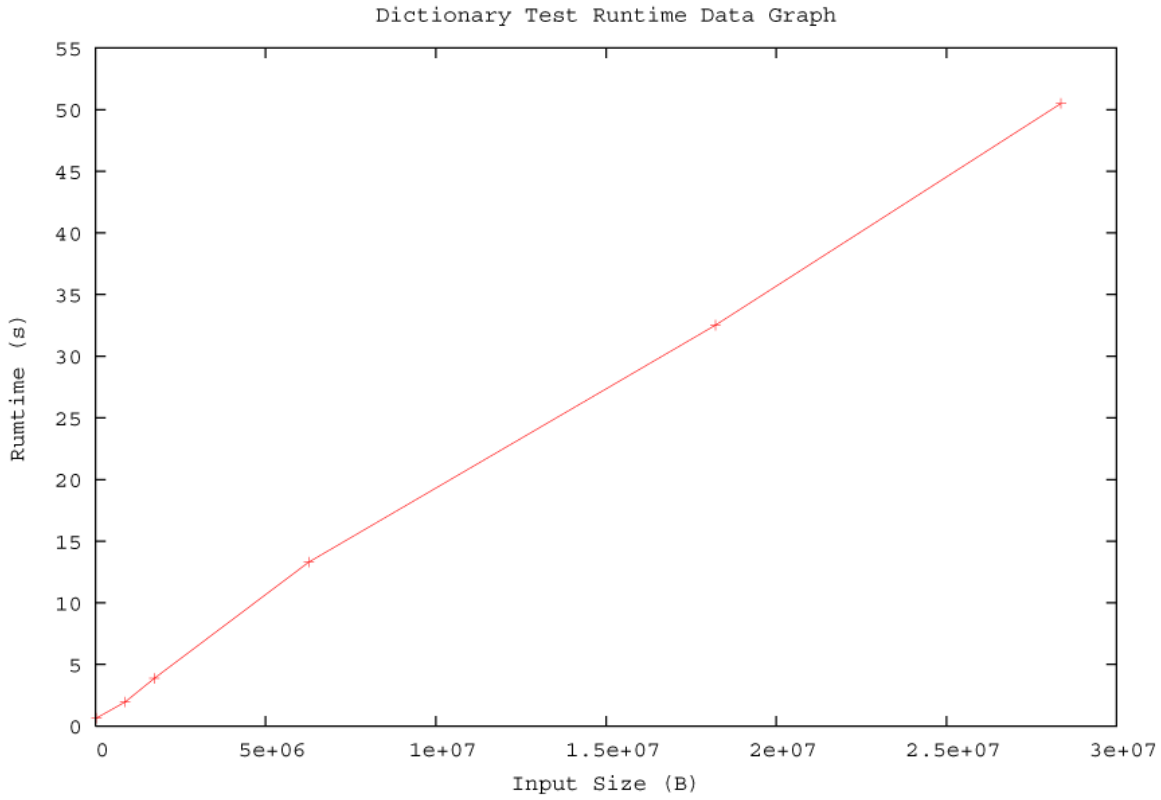
Size (B)	Words	Runtime (s)
349,022	62,023	1.94
1,056,104	190,695	3.23
2,112,208	381,390	5.15
7,563,408	1,295,826	14.58
21,256,973	3,046,112	33.80
33,107,752	4,774,404	51.77

5.8 Cracking Functionality

The algorithm used to crack the Vigenère cipher with a periodic key is the most complex algorithm in this project as it involves all of the other algorithms discussed previously . Although the program allows for three modes when cracking the Vigenère cipher, its runtime is dominated by the Kasiski test which is $O(N^2)$. The first mode uses the Friedman test to estimate a window of likely key lengths, and then tries each key length to find the one whose average *IC* is closest to that of the English language. This does not use the Kasiski test, and therefore its runtime is $O(N)$. The second mode uses the Friedman and Kasiski tests to estimate the key length and examines repeated substrings to find the most likely key length near the one estimated by the Friedman test. The third mode combines both of the previous two modes, and shows the results of both for comparison.

Figure 5.8 below shows the runtimes for all three modes and asymptotes arbitrarily selected to illustrate the algorithms' performance. It was found that all three modes agree with

Figure 5.7: Dictionary Test Runtime Data Graph



their theoretical runtimes, but fell increasingly below them as N increased. This is due to the nature of the English language. The likelihood of finding substrings of increasing length increases more rapidly with a smaller N. Analysis of approximately one million words yielded an average word length of 4.5 characters. It follows that the chances of even a long text containing a repeated 200 character pattern (which on average would contain 45 words) are quite low, whereas in smaller files the chances of finding additional shorter repeated patterns such as `THE` grows closer to the $O(N^2)$ asymptote due to the relatively short length of most words.

It is important to note that while modes 2 and 3 execute in $O(N^2)$ time, mode 1 executes in $O(N)$. It would seem that this efficiency might come at the cost of effectiveness, but the opposite was found to be true. For files of at approximately 7000 characters enciphered with a key of ten characters or less, the Friedman test was the only method that consistently yielded

the correct key. The Kasiski test failed in a small portion of the experiments. This is most likely due to the occurrence of long patterns that do not result from the encipherment of the same plaintext at the same place in the key. This could produce factors in the best-fit key length that are not in the actual key length, resulting in an incorrect best-fit key being produced by the Kasiski test.

Table 5.8: Cracking Functionality Runtime Data

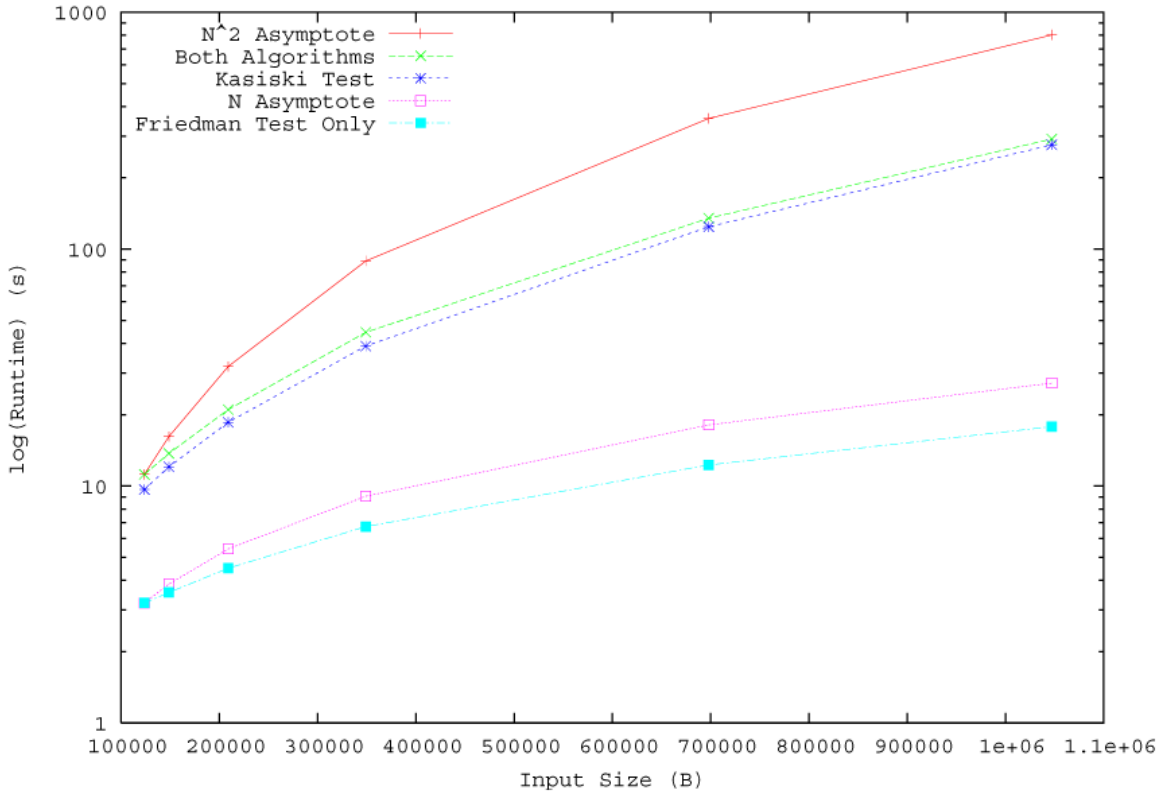
Size (B)	**Fried (s)**	**Kas (s)**	**Both (s)**
123,731	3.21	9.67	11.20
148,673	3.55	12.05	13.72
208,975	4.50	18.56	21.02
349,022	6.74	38.98	44.55
698,044	12.27	124.18	135.17
1,047,066	17.80	275.66	291.81

5.9 Chapter Summary

Determining the theoretical runtime of an algorithm is important, but perhaps more significant is how the algorithm actually performs. The Big-O runtime is an indication of the worst-case asymptotical upper bound, so a comparison with experimental values is useful to determine how close to the Big-O runtime an algorithm will perform with various inputs.

It was found that all of the algorithms are bounded by their theoretical Big-O runtimes, however some performed more closely to the bound than others. The algorithms for preparation, encipherment, decipherment, cracking multiple shift ciphers, the Friedman test, and the dictionary test all ran in $O(N)$ time. The dynamic programming implementation of the Kasiski test was bounded by $O(N^2)$, and the brute force algorithm was bounded by $O(N^3)$, but both diverged farther and farther from their asymptote as N increased. This is because the likelihood of finding increasing repeated patterns of increasing length tapers off as N grows.

Figure 5.8: Cracking Functionality Runtime Data Graph



Chapter 6

Conclusion

People have kept secrets since the dawn of time; employing ciphers is one means of doing so. Originating in the early 14^{th} century, the Vigenère cipher is a polyalphabetic substitution cipher based on the tableau in Section 2.2. Although there are many ways to form a key for encipherment, frequently a repeated (periodic) key is used.

This thesis has shown that if a periodic key was used, through a series of tests the Vigenère cipher can be broken with relative ease. Using only the Friedman test a ciphertext can be cracked in $O(N)$ time, and using the method which involves both the Friedman test and the Kasiski test a ciphertext can be cracked on $O(N^2)$ time.

The Kasiski test not only performed more slowly but was less effective than the Friedman test alone. While the Friedman test method consistently yielded the correct keyword, using the Kasiski test was less dependable. This is likely due to the possibility that long repeated substrings do not occur purely coincidentally.

Although it is both slower and less effective at cracking the Vigenère cipher than the Friedman test alone, the Kasiski test is interesting for a number of other reasons. Unlike the Friedman test, it is not a series of clear mathematical formulas. The abstract nature of the algorithm means that there are a number of possible ways to implement the data collection and analysis

processes, each of which will potentially alter the runtime and the suggested key length.

The dynamic programming algorithm developed for this project was tested against a brute force algorithm, and was found to be considerably faster and more efficient with memory usage. For a file of $6,389$ characters (about 3 pages) the brute force algorithm took slightly under six minutes to process, while the dynamic programming algorithm finished processing in under one half of a second. This is a almost a 700 times improvement in runtime for a relatively small N, demonstrating the benefit of dynamic programming algorithm design.

Additional study could be conducted into the effect of cache size on the runtimes of encipherment and decipherment via the tableau and computational methods. The size of the initial array used in the dynamic programming implementation of the Kasiski test could also have profound effects on the runtime, particularly for files with many shorter repeated substrings. An examination into the effect of changing the initial substring length could present interesting results.

Another area of experimentation that was not conducted is determining what the minimum necessary input file length is to the Kasiski and Friedman tests before they are effective, and how these sizes vary depending on various factors. Potential variables would include key length, particularly in relation to the file length, and the degree of patterns in the key and file. Examining the results when a long key consists of a smaller key repeated once with a single letter changed or how the number and length of repeated substrings in a file effect the amount of data that needs to be examined before the key length can be determined might prove to an interesting study in how to form a relatively strong periodic key.

The tests to determine how various factors effect the amount of data that must be considered before the key length can be reliably determined were not conducted due to time and complexity. Coupled with the work that this thesis project undertook, this would not have been a realistic goal. It remains an intriguing question, which begs to be answered by further research.

Pattern-matching algorithms such as the Kasiski test are not only interesting in terms of algorithm optimization, but also in their applications. Immediate applications of this project include additional research into the field of cryptography. Further optimization of the Kasiski test could be explored, and modules could be designed to handle the cracking of other ciphers. These might include a variant of Vigenère where each row of the tableau has been jumbled to counter the ease of cracking the cipher once the key length is known. As each row would still be a monoalphabetic cipher, the Friedman test could be used to determine the key length, and cracking would have to be explored from there. Transposition ciphers also pose interesting problems, and are not considered in this project. If one were paired with a substitution cipher such as Vigenère, this project could prove a useful framework from which to begin the cracking process.

The amount of data available to be analyzed has been growing exponentially in the last few years. As such, string analysis and pattern-matching algorithms such as the Kasiski test, especially algorithms that match sequences that are unknown prior to runtime, are directly applicable to the fields of data mining and Computational Biology [4, 5], which are two of the fastest growing fields in Computer Science. DNA sequencing is merely one of the practically limitless applications of pattern extraction algorithms.

This thesis has conducted an exploration into the cracking of the Vigenère cipher. It has shown that although the cipher is polyalphabetic, the periodic nature of its key proves to be a crucial weakness that can be exploited to crack a file enciphered with the Vigenère cipher in linear or $O(N)$ time. The Kasiski test runs in $O(N^2)$ time, but is of particular interest because of its parallels to solving current problems. Although this thesis centers on a dead cipher, its focus on algorithm efficiency and dynamic programming algorithms through the examination of the Kasiski test are widely applicable to problems that Computer Scientists are faced with today.

Appendix A

Functionality

Below is a brief description of how to execute the program, with a discussion of what each action does. This handles preparation of a file, encryption, decryption and cracking of the Vigenère cipher.

A.1 How to Run the Program

```
Usage: vigenere [ACTION] [MODE] [PASSPHRASE] [FILENAME]

ACTION is either '-e' for Encode, '-d' for Decode, '-c' for
        Crack, or -p for Prep. For -c or -p, the passphrase
        is omitted.

MODE is an integer, starting at 1, that defines which method
        of operation will be used for the given function. For
        -p MODE is 1 (include whitespace) or 2 (exclude
        whitespace). For -e and -d, mode 1 uses the tableau
        method and mode 2 uses the computational method of the
```

```
        vigenere cipher. For -c, mode 1 is the Friedman test
        only, mode 2 uses the Kasiski (and Friedman) test, and
        mode 3 is the results from both.

PASSPHRASE is a series of letters used to encode or decode
        the specified file.

FILENAME is the full or relative path and filename of the ASCII
        text file to be operated upon.
```

A.2 Preparation, -p

This project allows for preparation of input files and the dictionary to be used. The preparation of dictionaries is performed by a different executable, discussed in Appendix B.2.3. Preparation of input files strips all non-letters (excepting whitespace with mode 1), and casts all letters to uppercase. This way all output (from cracking or decipherment) can be compared against it's respective input easily using the `diff` command. This algorithm runs in $O(N)$ time. For further runtime analysis, please see Section 5.1.

A.3 Encode/Decode, -e/-d

The processes of encoding and decoding follow the algorithms laid out in Section 2.2.1. Both of these functions require that the input text file contain only uppercase letters and whitespace. Both encoding and decoding run in $O(N)$ time. For further runtime analysis, please see Sections 5.2 and 5.3.

A.4 Crack, -c

This is the heart of the project. It follows the algorithm described in Chapter 2 to crack the Vigenère cipher, and can be executed using the `-c` option as described in Appendix A. This will read in a file that has been encrypted with the vigenere cipher, and find the best-fit password. If the file contains whitespace, it will check the result to see if it is likely in English. The runtime is $O(N^2)$ if the Kasiski test is used, otherwise it is $O(N)$. For further analysis, please see Section 5.8.

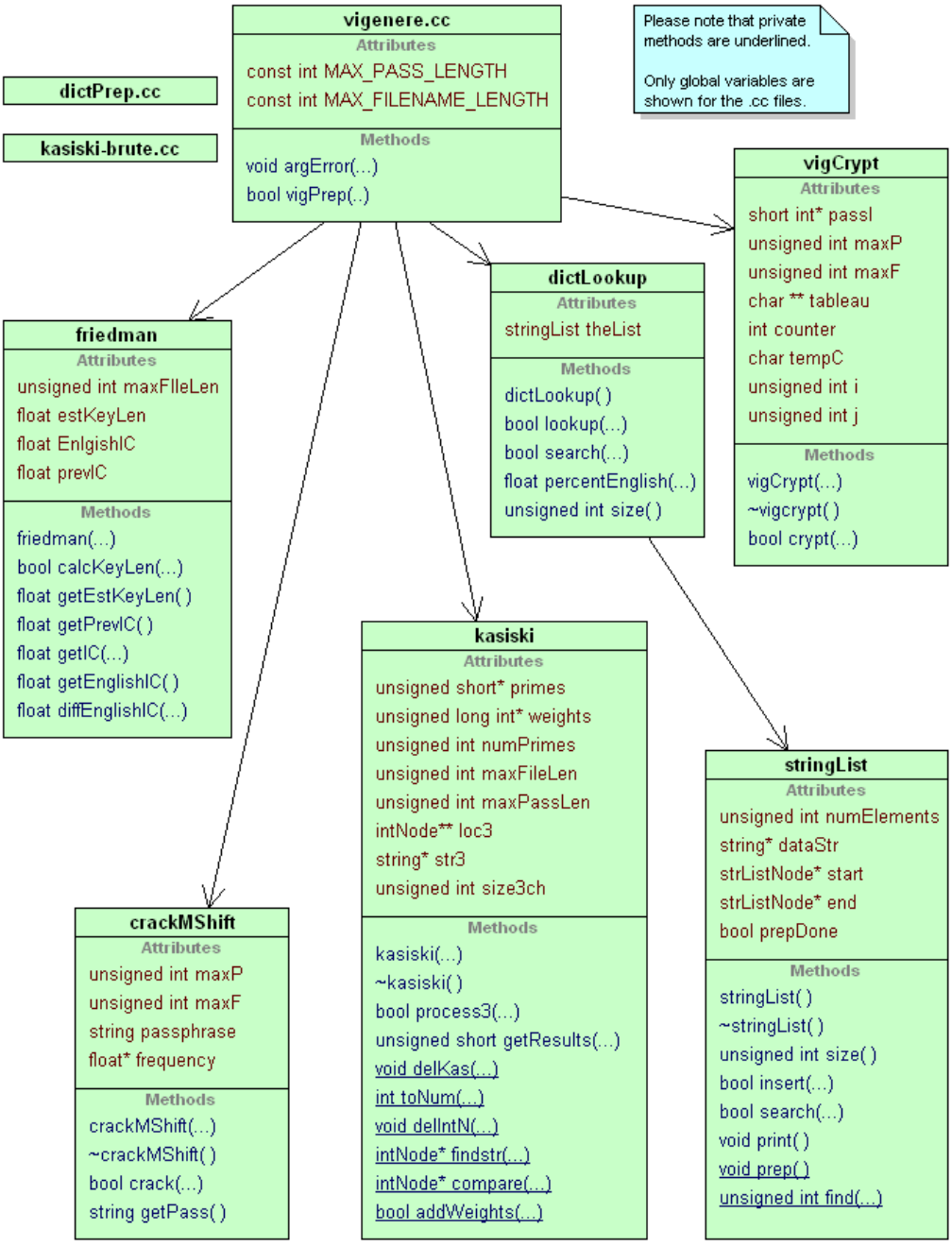
Appendix B

UML and Class Discussion

UML, short for Unified Modeling Language, is a way for Computer Scientists to show the structure of and relations between different classes in a programming project. The UML diagram for this project can be found below along with a list of the classes and drivers involved in the project and brief descriptions of what each does. The UML diagram was constructed using UML Sculptor [15].

B.1 UML

Figure B.1: Project UML Diagram



B.2 Drivers

These files are used to build the three executable files for this project: `vigenere`, `kasiski-brute` and `dictPrep`.

B.2.1 vigenere.cc

This is the driver for the main portion of the program. It supports encryption, decryption, and cracking of the Vigenère cipher as well as preparatory functionality which eliminates all illegal characters. Its usage is detailed in Appendix A. This program is bounded by the worst-case, which is cracking with the Kasiski test, resulting in a runtime of $O(N^2)$.

B.2.2 kasiski-brute.cc

This file contains a brute-force algorithm for the Kasiski test written by Eric Reed, Bucknell University class of 2006. It blindly checks for repeated substrings of increasing length until none are found. This is done by comparing every substring with each consecutive substring at most $(N/2)+1$ times, resulting in a runtime of $O(N^3)$. It can be compiled using the `make all` command and must be run by piping the input file contents into it as follows, where `[FILENAME]` is the file to be examined by the Kasiski test:

```
kasiski-brute < [FILENAME]
```

B.2.3 dictPrep.cc

This file is used to prepare dictionary files (ASCII files containing a list of words) by removing all words that contain characters that are not standard American English letters. It uses a single pass through the input file, and hence runs in $O(N)$. Although the included dictionary has already been cleaned, dictPrep can be compiled using the `make all` command, and

can be executed with the command `dictPrep [FILENAME]` where `[FILENAME]` is the dictionary to be prepared, and the output will be `FILENAME.dictPrep`.

B.3 Classes

B.3.1 crackMShift

This class divides the ciphertext into L groups, and uses frequency analysis to determine what shift was used to encipher each group. This information is the used to determine the key used to encipher the file. As L is bounded by a constant, the process executes in $O(N)$ time. The file can then be decrypted, and the Vigenère cipher is effectively broken.

B.3.2 dictLookup

This class builds a dictionary (searchable list of words) to be used to determine whether a file is likely to be in English or not. It extends the functionality of stringList. As the number of entries in the dictionary is constant, the process of searching for every word in an input file is bounded by $O(N)$.

B.3.3 friedman

This class performs the various functionalities of the Friedman test in $O(N)$ time. This includes determining ICs, calculating the estimated key length, and being used to find the best-fit key length in a range of values.

B.3.4 kasiski

This class performs all functions of the dynamic programming implementation of the Kasiski test, as well as combining its results with the result of the Friedman test to determine the best-fit

key length. It is bounded by $O(N^2)$.

B.3.5 stringList

This class allows for fast creation, $O(D)$, and searching, $O(\log D)$, of a series of consecutively input strings, assumed to be inserted in lexicographical order. As D is the size of the dictionary and hence constant, these both simplify to $O(1)$. This also assumes that all insertions will be made before a search is performed. Removal is not allowed, but could be implemented in the future if necessary.

B.3.6 vigCrypt

This class handles encryption and decryption of files via the Vigenère cipher. It allows for both the Tableau and Computational methodologies as discussed in Section 2.2, both of which execute in $O(N)$ time.

Appendix C

Source Code and License

The source code for this project can be downloaded from this URL:

`http://mattberntsen.net?page=thesis`

The entirety of this project is licensed under the Open Software License v. 2.1 which is distributed with the source code and can be obtained at either of these URLs:

`http://www.opensource.org/licenses/osl-2.1.php`

`http://mattberntsen.net/code/thesis/License.txt`

Appendix D

Programming Environment and Required Software

This project was developed on the Sun Solaris 9 Operating System [16]. There is one call to `system()` in vigenere.cc, which executes a `rm` command, which requires the program to run on either UNIX or Linux. Compilation was done using GCC Version 3.3 [14], and the project was maintained using CVS Version 1.11.17 [17].

Appendix E

Definitions

Array A method of storing a known number of variables contiguously in memory such that each variable contained inside can be accessed by name in $O(1)$ time.

Autokey Cipher A cipher that incorporates the plaintext into the key. This uses a key that consists of a short priming key concatenated with the plaintext.

Binary Search A method of searching a sorted array that halves the area of the array being examined in every iteration. Searches run in $O(\log N)$ time.

Cache Extremely fast memory used in a computer to store frequently accessed data or instructions.

Cipher A method by which plaintext is reversibly transformed into a less intelligible ciphertext.

Ciphertext The text that results from encipherment of a plaintext.

Encryption A method by which information (text or data) is transformed such that it is reversibly less meaningful.

Frequency Analysis A tool in cryptanalysis that is used in cracking monoalphabetic ciphers. The known frequencies of letters in the English language (Found in Table 2.1.) are compared with the frequencies of the letters in the ciphertext. If a shift can be found where the two sets of frequencies match closely, the cipher used is likely to be an additive cipher with the indicated shift. If not, other tools must be used to crack the cipher.

Key A value used by a cipher to encipher plaintext to ciphertext and decipher ciphertext to plaintext. Although these can take on many forms, keys in this project will be strings of contiguous letters such as "`ABC`" or "`BUCKNELL`".

Keyword A key that consists only of letters.

Linked List A dynamic method of storing an unknown number of variables in memory. Only the start point and the next variable in the linked list are known at by any node, so accessing any specific variable inside will take $O(N)$ time.

Passphrase See "Key".

Plaintext A text in ordinary English to be enciphered.

Mod or Modulo A mod B is defined as the remainder after A is evenly divided by B.

Monoalphabetic Cipher A cipher in which one letter of plaintext is uniformly mapped to one and only one letter of ciphertext [7, 12].

Polyalphabetic Cipher A cipher in which the letters are mapped one-to-many from plaintext to ciphertext based on some algorithm [7].

Runtime A way of describing how the time necessary for a program or function to run changes in relation to the size of the input. The Big-O Notation describes an asymptotical upper bound for the runtime as N approaches infinity. Formally, a function is $O(g(N))$ if and

only if for its runtime F there exist positive constants C and N_0 such that $0 \leq F \leq Cg(N)$ for all $N \geq N_0$. For the duration of this thesis N will denote the size of the input to a given function or algorithm, which is often the number of characters in a file being analyzed.

Shift Cipher Formally known as an Additive Cipher, shift ciphers work by mapping each plaintext letter to a ciphertext letter a fixed number of positions beyond it in the alphabet. For more information, please see Section 2.1.

Substitution Cipher A cipher that takes each plaintext letter and through some algorithm, either monoalphabetic or polyalphabetic, enciphers it into a ciphertext letter.

Sum Square Error Method A method of computing a the goodness of fit of an additive cipher. Given the letter frequencies F_1 to F_{26} found in Section 2.1.1 the frequencies G_1 to G_{26} are calculated for each of the possible 26 shifts. The error E is then calculated as follows for each possible shift. The shift with the lowest error is the best fit.

$$E = \sum_{i=1}^{26} (F_i - G_i)^2 \tag{E.1}$$

Text Cipher A method of concealing information that works directly on the letters (text) as opposed to their data representation.

Transposition Cipher A cipher that seeks to break the relationships between letters through an algorithmic reordering.

Vigenère Cipher Please see Section 2.2.

Whitespace Any character or series of characters that are displayed as empty space. This includes spaces, tabs and line termination characters.

Appendix F

Acknowledgements

I would like to thank my advisor Professor Richard Zaccone as well as Professor L. Felipe Perrone for their continued support and confidence, as well as for agreeing to sit on my Review Board.

I would like to thank the Honors Council and Professor Matt Higgins, the third member of my Review Board, for allowing me the opportunity to work on this project.

I would like to thank Professor Gary Haggard and Bucknell's Computer Science Department for their commitment to their students and the education that we have all received at Bucknell.

I would like to thank Eric Reed, Bucknell '06, for allowing me to use his string hashing code and to test my implementation of the Kasiski test against his. I would like to thank Aaron Schwager for the use of his string hashing function. I would also like to thank the Engineering Computing Support Team (ECST) for accommodating my requests with the utmost efficiency.

Lastly, I would like to thank my family and friends for their tolerance while I worked on this project.

Bibliography

[1] Pfleeger, Charles P. and Pfleeger, Shari L. *Security in Computing*. Third Edition. Pearson Education, 2003.

[2] Lewand, Robert E. *Cryptological Mathematics*. The Mathematical Association of America, 2000.

[3] Singh, Simon. *The Code Book: The Evolution of Secrecy from Mary, Queen of Scots to Quantum Cryptography*. Doubleday, 1999.

[4] Waterman, Michael S. *Introduction to Computational Biology*. Chapman & Hall, 1995.

[5] Gusfield, Dan. *Algorithms on Strings, Tree, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[6] Gaines, Helen F. *Crpytanalysis: A Study of Ciphers and Their Solution*. Dover Publications, 1939.

[7] Menezes, Alfred J, van Oorschot, Paul C. van and Vanstone, Scott A. *Handbook of Applied Cryptography*. CRC Press, 1996.

[8] Kasiski, Freidrich W. *Die Geheimschriften und die Dechiffrierkunst*. 1863.

[9] Vardeman, Stephen B. and Jobe, J. Marcus. *Basic Engineering Data Collection and Analysis*. Duxbury, 2001.

[10] Sedgewick, Robert. *Algorithms in C++ Parts 1-4*. Addison-Wesley, 1998.

[11] Open Software License v. 2.1, Accessed February, 10 2005.

`http://www.opensource.org/licenses/osl-2.1.php`

[12] Oxford English Dictionary Online. Accessed October 10, 2004.

`http://dictionary.oed.com`

[13] Project Gutenberg, Accessed February - April 2005.

`http://www.gutenberg.org`

[14] GNU Compiler Collection. Accessed February 19, 2005.

`http://gcc.gnu.org/`

[15] UML Sculptor, Accessed April 4, 2005.

`http://sourceforge.net/projects/umlsculptor/`

[16] Sun Solaris. Accessed February 19, 2005.

`http://www.sun.com/software/solaris/`

[17] Concurrent Versions System. Accessed February 19, 2005.

`http://www.gnu.org/software/cvs/`