Decrypting the Vigenere Cipher

Andrew Huang

Abstract. In this paper, I explore decrypting the Vigenere Cipher through the Metropolis Hastings algorithm, a Monte Carlo markov chain method. I use the algorithm to crack ciphers of known and unknown length, and compare the efficiency to an optimized brute force solution.

Keywords and phrases: Vigenere, cipher, Metropolis Hastings, monte carlo, markov chain

1. Introduction

The Vigenere Cipher is a form of a polyalphabetic substitution cipher that uses a secret “codeword” to encrypt text. Based on the letters of the codeword, one implements interwoven Caesar ciphers to encrypt each letter. For example, if the plaintext to be encrypted is “HELLO” and the codeword is “JOE”, the “H” will be encoded by a Caesar cipher with shift “J”, the “E” with “O”, the “L” with “E”, the next “L” with “J”, and the “O” with “O”, so “HELLO” aligns with “JOEJO”. This yields “QSPUC” as the ciphertext.

Developed in the 16th century, it was thought to be “indecipherable” for many years. Finally, in the 19th and 20th centuries, cryptographers were able to crack this code through frequency analysis[1]. In this paper, I compare decrypting the cipher through the Metropolis Hastings algorithm with more traditional methods to see if the Metropolis Hastings algorithm is more efficient than regular, brute force methods.

To decrypt the Vigenere Cipher, I utilized the textbook’s method computing a score based on inter-letter frequencies. I decided to “guess” a codeword solution, similar to how the textbook “guesses” a substitution function. The message I used to decrypt is attached in Appendix A (the same one used in the textbook), and the codewords I used were “lemon” for length 5, and “lemonextratext” for length 14. These were picked because the example on Wikipedia used “lemon” as the sample codeword.

2. Vigenere Cipher with known length

When the length of the codeword is unknown, decrypters must guess the length of the codeword, making it much more difficult to crack. I first attempted cracking the cipher with the length of the codeword known.

2a. Brute Force Algorithm

The first method I used was simply brute forcing the decryption. This can be done very easily; if the codeword length is n, one can simply iterate through the 26^ncombinations for the codeword. I wrote a program in R (See Appendix B for the code), based on decode.R from the textbook for substitution algorithms[2], to implement this algorithm. However, this is has a runtime of$O(26^n)$, meaning that 1 additional input letter results in a 26x increase in runtime. By extrapolating the data in Table 1 below, a simple 5-letter codeword would take 3.68 hours to decrypt. This method is infeasible in practice. In each of the tables in this paper, I utilized 5 trials. The raw trials are attached in Appendix H (along with all other raw trials). The pseudocode for this is below.

```
brute_force()
  for each n-combination of 26 letters
    Compute score for each word
  Pick word with highest score
```

Table 1: Brute Force Decryption Times

Codelength	*Mean (s)*	*Stdev (s)*
1	0.6354	0.0350
2	18.0988	1.1722

2b. Optimized Brute Force Algorithm

To improve on this brute-force algorithm, one can perform a slight optimization. I decided to iterate over each letter and select the best letter at each position. This means I pick the letter with the best score in the first index, keep it, and then iterate over the next indices such that by the end, the codeword is finalized. For instance, if the codeword is “lemon” and the algorithm is initialized to “aaaaa”, the next iteration will be “laaaa”, and then “leaaa”, “lemaa”, “lemoa”, and finally “lemon”. This algorithm runs in $O(26n) = O(n)$time, a significant improvement (n is the length of the codeword). I utilize this optimized brute force algorithm as my “baseline” of comparison with the effectiveness of the Metropolis Hastings algorithm to decrypt the cipher. See Table 2 on the next page for a comparison with the Metropolis Hastings algorithm, and Appendix C for the raw R code. From Appendix C onwards, I will only attach the lines of code that are changed between algorithms (i.e. the function for encryption and decryption do not change, so I will not include those).

```
brute_force_optimized()
  for each index, i, in codeword length
    1. Compute score with each letter in
    index i
    2. Pick letter with highest score
```

Note that this algorithm is not guaranteed to work for every message and codeword combination. Theoretically, a greedy approach to picking letters may not yield in the final decrypted solution; however, it is mathematically unlikely that at any point in the algorithm, the greedy choice is not the correct one. To circumvent such a possibility, one can brute force the first three letters, at which point it becomes almost mathematically impossible for the greedy letter choice to be incorrect, based on inter-letter frequencies. However, this modification was not needed, as the message was decrypted perfectly for every trial.

2c. Metropolis Hastings Algorithms

To implement the Metropolis Hastings algorithm to decrypt the ciphertext with known codeword length, it is fairly straightforward. The idea is to start with the identity word, “aaaaa” (if the codelength if 5), and randomly select an index to replace it with a random letter. At each iteration, the acceptance function of new word, f^*, and old word,f, simplifies to

$$a(f, f^*) = \frac{\pi_{f^*} T_{f^*,f}}{\pi_f T_{f,f^*}} = \frac{\pi_{f^*}}{\pi_f} = \frac{score(f^*)}{score(f)}$$

since the transition matrix is symmetric (it is equally likely to get from any five letter word to another, since the index and letter is selected randomly). The pseudocode is below.

```
mcmc_fixed_length()
  while(message != decoded)
    i <- random index
    l <- random letter
    f*<- replace(f, i, l)
    U <- runif(0, 1)
    if U < a(f,f*), f <- f*
```

I designed the algorithm to terminate when the decrypted message matched the original (see Appendix D for raw R code). However, without the original message this is not possible. Thus, I also ran the algorithm to output the message after a fixed number of trials (threshold of 800 for length 5 and 2000 for length 14). These thresholds were selected from trial and error. It is evident that these two algorithms are slower than the brute force method; one big part of the runtime overhead is the usage of a map to store already seen functions and their score. I implemented another modification of this algorithm without the map. The results are compiled in Table 2 on the next page, for a 5 letter codeword and a 14 letter codeword (see introduction). Algorithm 1 is optimized brute force, 2 is MCMC with decrypted termination clause, 3 is MCMC with fixed iterations, and 4 MCMC is without a map (and decrypted termination clause).

Table 2: Known code length decryption (codeword: "lemon")

Algorithm	*Mean (s)*	*St. Dev. (s)*
1 (brute)	3.0726	0.0409
2 (decrypted)	7.8800	2.9631
3 (iterations)	7.6526	1.6459
4 (no map)	10.6829	1.4607

Table 3: Known code length decryption (codeword: "lemonextratext")

Algorithm	*Mean (s)*	*St. Dev. (s)*
1 (brute)	9.4347	0.3597
2 (decrypted)	34.5787	8.0361
3 (iterations)	31.6178	6.4682
4 (no map)	30.8271	13.2739

From these results, it is clear that the brute force algorithm, regardless of codeword length, is optimal to the MCMC algorithm. For a short codelength, since it is highly probable that one will see repeated words, using a map is preferable in the MCMC implementation. For a long codelength, decrypted, iterations, and no map all had relatively similar times. Most notably, no map had a very high standard deviation. This makes sense as iterations requires every trial to run the same number of iterations (thus deviation is small), while when comparing decrypted vs. no map, no map does not cache past results, resulting in a high standard deviation. The algorithm may discover the codeword quickly without seeing many repeated words, or may see many repeated words and slow down the runtime.

To see if the brute force algorithm is significantly faster than the MCMC algorithms, I then conducted one-tailed, heteroscedastic t-tests. The reason for heteroscedasticity is that the distributions have different variances. The p-values are listed below, in Tables 2a and 3a, showing that the brute force algorithm was significantly faster than all MCMC variations if we use a critical value of 0.05 (although, if we use one of 0.01, there are some that have no significant difference).

Table 2a: 5-letter p-values

Algorithm 1	*Algorithm 2*	*p-value*
1 (brute)	*2 (decrypted)*	0.01110
1 (brute)	*3 (iterations)*	0.00169
1 (brute)	*4 (no map)*	0.00015

Table 3a: 14-letter p-values

Algorithm 1	*Algorithm 2*	*p-value*
1 (brute)	*2 (decrypted)*	0.00109
1 (brute)	*3 (iterations)*	0.00076
1 (brute)	*4 (no map)*	0.01133

In this section, I did not attempt to crack a longer codeword, simply because the results for all algorithms scale similarly in length of codeword. I will attempt to decrypt a longer codeword when the codeword length is unknown.

3. Vigenere Cipher with unknown length

An optimized brute force algorithm on a known length codeword will perform better than MCMC simply because given the length of the codeword, there are only $O(n)$words to try. However, the interesting question is when the codeword length is unknown, which is the likely case in a real-world situation. In this case, there are $O(m)$possible lengths to try, where m is the length of the message. At every length, one must also try every possible word, yielding a $O(m^2)$algorithm. In computer science, this is pseudo polynomial time, since m could be any function of n (polynomial, exponential, etc.). In this section, I explore if this poses a problem to the brute force or MCMC algorithms. Our message has 476 non-character spaces, significantly greater than 5 or 14. Since the codeword could be of any length from 1 to 476, a brute force algorithm will have to try every possibility.

3a. Optimized Brute Force Algorithm

Using the optimized brute force algorithm outlined in section 2b, the algorithm I designed will simply guess each possible length, from 1 to the length of the message. At each length, i, it will run the algorithm in 2b to get an optimal score. The length with the best optimal score is likely to be the codelength, and the optimal word is likely to be the codeword.

Theoretically, to be positive that the best codeword was attained, one must iterate through every possible length for the codelength. However, in practice, the actual codeword will have a score that is significantly higher than the optimal codeword at any other length. For the codeword "lemon", which has length 5, I ran the optimized brute force algorithm for lengths 1 to 8. The data, in Table 4 below, shows that the optimal scores of other codelengths are barely over 4100, while the score of "lemon", the correct codeword, was almost at 6000. While these numbers are definitely a case-to-case basis, a termination clause for a general optimized brute force algorithm of unknown length can be when the optimal score of a length has a 10% increase from the best score seen so far. This reduces the algorithm's runtime from $O(m^2)$to $O(n^2)$, a significant change from pseudo polynomial time to polynomial time. See Appendix E for the code.

Table 4: Optimal Score vs. Codelength

Guessed length	*Optimal Score*
1	3928.699
2	3966.400
3	4008.996
4	4122.074
5	**5836.555**
6	4025.324
7	4062.304
8	4124.391

For this brute force algorithm, aside from testing codewords of length 5 and 14, I tested one of length 50 - "lemonthereasonipickedlemonwasbecauseitwasanexample". I did this because the algorithm runs in $O(n^2)$time. This means that the longer the codeword, the longer it must run. I wanted to see if the results were proportional to n^2. The results are shown below in Table 5.

Table 5: Optimized Brute Force (unknown codeword length)

Codeword	*Mean (s)*	*St. Dev. (s)*
5 letters	10.2915	0.3267
14 letters	77.8729	1.7434
50 letters	875.9871	25.3011

The results show that the average time to run the algorithm was proportional to the length of the codeword squared, roughly 0.35-0.40x the square of the codeword.

3b. Metropolis Hastings Algorithm

For the Metropolis Hastings algorithm, deciphering an unknown length can be done in various ways. As long as the transition matrix of the markov chain is symmetric, the acceptance function will not require complex calculations, and simply be a ratio of the two scores. However, the choices come in deciding a transition matrix such that$T_{f^*,f} = T_{f,f^*}$, so the acceptance function simplifies to:

$$a(f,f^*) = \frac{\pi_{f^*}T_{f^*,f}}{\pi_f T_{f,f^*}} = \frac{\pi_{f^*}}{\pi_f} = \frac{score(f^*)}{score(f)}$$

In this section I will discuss various transition matrices in cracking the Vigenere cipher.

Note that in all Metropolis Hastings implementations, the algorithm runs independent of the length of the codeword. Since an "honest" MCMC algorithm should allow transitioning from any word length to another, it should not depend on the length of the codeword but rather the possible lengths of the codeword - this is always an $O(m)$time algorithm. I will discuss the complications that arise when using a "pure" MCMC algorithm.

3bi. Random Transition Matrix

The first matrix that came to mind was an absolutely random transition matrix. This means that any word can transition to any other word - the length is random, and the letters are randomly generated. However, this is a poor decision. Even though $T_{i,j} = T_{j,i}$for any words i, j, this value is equal to $T_{i,j} = \frac{1}{O(26^m)}$, since there are that many words. Essentially, this algorithm would work, but would require the algorithm to "guess" the correct word. The likelihood of this is very slim, and would take too long to run.

3bii. Random Transition Matrix

I tried to implement an algorithm that keeps its "progress" from past iterations, similar to the substitution cipher MCMC solution provided in the textbook, where one swaps out letters to generate new codewords. It improves upon the words seen before and swaps out a letter at a time.

However, with an unknown length, this is very difficult and I was unable to think of an algorithm that could utilize MCMC to guess the length of the codeword. I initially thought about using an algorithm that would not only swap out a letter, but also manipulate the length of the codeword. The algorithm would maintain a "running codeword" of length m, the length of the message, and at each iteration, pick a random spot to "stop" the codeword and pick a random letter to swap out. For instance, if the message is length 5, the algorithm will maintain a "running codeword" of length 5, i.e. "testx". At each iteration, the algorithm will decide to pick a random index to stop at and a random letter in the range (0, index) to swap out. In the case of "testx", it could pick index 4 and swap the letter at index 3 with "c". This would yield a codeword "tect". The reason for this is that it will manipulate both the length and the letters, and hopefully converge to the answer.

However, this transition matrix is no longer symmetric - and it's no longer a markov chain. *Any* algorithm that keeps a "running codeword" and changes the length at each iteration is no longer a Markov Chain. For instance, let's ignore the swapping of the letters for now. Going from "te" to "test" depends on what the letters at other positions of the "running codeword" is, and thus, prior iterations of the algorithm. I could not think of a way to solve this problem, so I decided to find ways to optimize and pick certain codeword lengths to work with, moving away from a pure MCMC algorithm. I was unable to figure out a reasonable algorithm without predetermining a code length.

4. MCMC Optimization

In this section, I will focus on optimizing the MCMC approach specifically, as this pertains the most to our class material. The second optimization in particular can be applied to the brute force solution, but this section will focus on optimizing MCMC only.

The problem with pure MCMC approaches to decrypting the cipher is that it is an $O(m)$time algorithm - the size of the message, or the sample space of the codeword length, is the bottleneck for the number of iterations it must run. Additionally, any algorithm that manipulates the length of a codeword is very difficult to implement in an MCMC format. Thus, making educated guesses, or at least some guesses about the length, will significantly improve the performance of the MCMC algorithm.

4a. Optimized MCMC

Similar to the optimized brute force algorithm we ran in section 3a, we can iterate over the numbers 1 to length of message to guess the length of the codeword. At each iteration, we can run an iteration-based MCMC algorithm from section 2b, and similar to how I performed the 10% termination rule in section 3a, terminate when a significant score improvement is seen. For each codeword length, I ran 150x the codelength iterations; i.e. for guessing a length of 4, I would run $4 * 150 = 600$iterations. I present the results below, in Table 6. I did not attempt to decode the 50 letter codeword since the algorithm performed significantly slower (Table 6a) than the brute force algorithm.

Table 6: Optimized MCMC

Codeword	*Mean (s)*	*St. Dev. (s)*
5 letters	20.1869	2.0737
14 letters	221.7014	15.9989

Table 6a: Brute Force vs. MCMC (unknown)

Number of Letters	*p-value*
5	0.00109
14	0.00003

Similar to Tables 2a and 3a, I ran one-tailed, heteroscedastic t-tests to see if the runtimes were significantly different, and they clearly are. This is because the MCMC requires going through $150 * i$ iterations for every i, instead of $26 * i$ for brute force. I attempted to optimize the algorithm by running $26 * i$ iterations and seeing if the score is significantly higher (and then deciding if to move on to the next i or to continue, if it is significantly higher), but often MCMC does not converge to a decent guess after $26 * i$ iterations, even if i is the correct codelength. See Appendix F for the raw R code.

4b. Kasiski MCMC

In 1863, a Prussian named Friedrich Kasiski discovered a method of guessing the codeword length. He noticed that strings of three letters that were repeated throughout the encrypted message usually corresponded to multiples of the codeword. For instance, if the three letter combination “xyz” is repeated multiple times throughout the ciphertext, it is very likely that these correspond to the same letters in the plaintext that were encrypted with the same part of the codeword. Thus, analyzing the factors of the distance between repeated strings of three characters almost always yielded the codeword[3]. For example, if the distance between the two “xyz” was 16 characters, the most likely codeword lengths would be out 1, 4, and 16.

When implementing this version, I calculated all factors of the distances between repeated three letter combinations. However, for the choices of codelength, I only picked the factors that appeared in at least a third of all distances (excluding the factor of 1). This allows me to test the most likely cases of the codeword length, but not leaving out too many candidates. Below, I show the results of the time required to process the factoring (Table 7a), and the time for the overall decryption (Table 7b). I have attached the code in Appendix G.

For this algorithm, I sorted the candidate lengths in increasing order, and set a cutoff of a 10% increase in optimal score. To recap what this means, if the candidate lengths are 1, 4, and 8, I will run the MCMC algorithm of fixed length for these three lengths. If any of these lengths yields a 10% increase between their optimal score and the previous best score, I will output the length and word that yielded the best results. The reason for first sorting the candidates is that there tends to be a very slight increase between score and guessed length. This can be rationalized since “lemo” is likely to yield better decryption attempt than “l” if the codeword is “lemon”.

Table 7a: Kasiski MCMC Factoring Time

Codeword	*Mean (s)*	*St. Dev. (s)*
5 letters	0.1089	0.0136
14 letters	0.1162	0.0112

Table 7b: Kasiski MCMC Runtime

Codeword	*Mean (s)*	*St. Dev. (s)*
5 letters	15.2244	1.4063
14 letters	76.1313	8.1845

It is pretty clear that the overall factoring time overhead is not a strong concern when running the algorithm, as it just depends on the length of the message. The runtime is very similar to that of an optimized brute force solution - while it tries less combinations, it still must go through $150 * i$iterations for each length, i. This means that combining Kasiski with the optimize brute force algorithm will likely yield the best results. However, this is the best algorithm for MCMC that I was able to attempt and run.

Note that Kasiski analysis is not guaranteed to converge to the correct answer - one case is if there are no repeated strings, for instance. If it does not, one would just continue to the Optimized MCMC outlined in section 4a. This was not needed for the codeword of length 5 and 14. However, for the codeword of length 50, there were two problems. First, the codeword is very long relative to the size of the message, so there were very few repeated sets of three letters. In addition, the string “lemon” is in the codeword twice, which artificially adds repeated strings in the ciphertext (these will not be in a multiple of the codeword length). The solution to this would be to take every factor (since 50 will likely only appear a few times, if any). Since I established that MCMC algorithms are inferior in runtime to the optimized brute force, it would be efficient and better to couple this method with the optimized brute force algorithm.

5. Vigenere, without spaces

To make it harder for decryption, people often separate the ciphertext into chunks of equal sizes. For instance, in plaintext form, “I want to eat” may be split into “Iwan ttoe at” if the chunks are of size 4[4]. This makes it more difficult to analyze, as spaces do not actually denote the stoppage of a word.

To decrypt this, I removed all spaced in the original message. While this will definitely negatively affect the score, (i.e. Iwant will score the transition from I and w, but these are not actually consecutive), but without complex analysis of what is likely to be a transition, it is impossible to ignore. Table 8 has the results of using the optimized Kasiski MCMC on the Vigenere without spaces.

Table 8: Vigenere, without spaces

Codeword	*Mean (s)*	*St. Dev. (s)*
5 letters	10.0996	1.8540
14 letters	63.1329	11.3020

Something that is surprising is that this was actually faster than the Kasiski MCMC on the actual message. The reason for this is likely because the message then shrinks in size, by a significant factor.

However, there was a quirk when running this algorithm. Instead of using a 10% increase in score, I had to use a 20% increase in score. This is because by removing the spaces, the scores of the letters were reduced by a significant factor. 10% of 2500 is a lot smaller than 4000, so the algorithm often would terminate prematurely when set to 10%.

6. Conclusion

I ran many algorithms to decrypt the Vigenere cipher. I first ran a brute force algorithm before developing an optimized brute force algorithm. For MCMC algorithms, I ran three variations of one algorithm, where I varied how the algorithm terminated. There was not much of a difference between these three variations.

Next, I moved onto unknown length algorithms. I tried iterating through all possible codeword lengths, but discovered it was optimal to stop if there was a 10% increase in score. I then attempted to make a pure MCMC algorithm to decrypt the Vigenere of unknown length, but found it to be too difficult. I then applied the optimization of stopping with a 10% increase in score to the MCMC. However, I researched and found the Kasiski analysis of Vigenere ciphers, which decreased the runtime of the MCMC algorithm for an unknown length.

In the end, the optimized brute force algorithm was always better than the MCMC algorithms. This is because for a certain length i, the brute force only needs to run $26 * i$ iterations, but the MCMC must run $150 * i$iterations to converge to the correct answer. Regardless, using the 10% increase in score and using the Kasiski analysis will optimize the runtime. Additionally, removing spaces may actually yield faster results.

Future work for on this topic would be to develop an actual, “pure” MCMC algorithm - one that does not predetermine codeword length. Also, other forms of analyzing possible codeword lengths that would work for codewords with repeated strings is a promising topic as well.

For this project, I worked alone and developed all the programs myself. I used the textbook’s decode.R for substitution ciphers as a backbone of my code, but did not use any other external resources. I would strongly encourage anyone to use the code to replicate my results!

7. Works Cited

[1] https://www.britannica.com/topic/Vigenere-cipher
[2] http://www.people.carleton.edu/~rdobrow/StochasticBook/Rscripts/decode.R
[3] https://inventwithpython.com/hacking/chapter21.html
[4] https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-Base.html

8. Appendix

A. Message

coincidences in general are great stumbling blocks in the way of that class of thinkers who have been educated to know nothing of the theory of probabilities that theory to which the most glorious objects of human research are indebted for the most glorious of illustrations edgar allen poe the murders in the rue morgue morpheus this is your last chance after this there is no turning back you take the blue pill the story ends you wake up in your bed and believe whatever you want to believe you take the red pill you stay in wonderland and i show you how deep the rabbit hole goes

B. Brute Force Code (Known)

```
# decode_vigenere_brute_force.R
# decode with brute force

message <- "coincidences in general are great stumbling blocks in the way of that class of
thinkers who have been educated to know nothing of the theory of probabilities that theory
to which the most glorious objects of human research are indebted for the most glorious of
illustrations edgar allen poe the murders in the rue morgue morpheus this is your last
chance after this there is no turning back you take the blue pill the story ends you wake up
in your bed and believe whatever you want to believe you take the red pill you stay in
wonderland and i show you how deep the rabbit hole goes"
codeword <- "lem"

setwd('/Users/andrewhuang/Documents/Sophomore2018/STAT433FinalProject')
mat <- read.table("AustenCount.txt",header=F)
logmat <- log(mat + 1)

# Computes the score of the decoded message using the given code
score <- function(code)
{
  p <- 0
  for (i in 1:(nchar(message)-1)){
    p <- p + logmat[charIndex(substr(code, i, i)),charIndex(substr(code, i+1, i+1))]
  }
  p
}

# ascii(char) returns the numerical ascii value for char
ascii <- function(char)
{
  strtoi(charToRaw(char),16L) #get 'raw' ascii value
}

# charIndex takes in a character and returns its 'char value'
charIndex <- function(char)
{
  aValue <- ascii(char)
  if (aValue == 32)
  {
    27
  } else
  {
    aValue - 96
  }
}

# Encrypts code according to codeword
encrypt <- function(code, codeword)
```

```
{
  out <- code
  index <- 1
  # for each character in the message, encode it according to the codeword
  for (i in 1:nchar(message))
  {
    charInd <- charIndex(substr(code,i,i))
    codewordInd <- charIndex(substr(codeword,index,index))
    if (charInd < 27)
    {
      # change the ith character to the character determined by the codeword
      substr(out,i,i) <- rawToChar(as.raw(((charInd + codewordInd - 2) %% 26 + 97)))
      index <- index + 1
      if (index > nchar(codeword)){
        index = 1
      }
    }
  }
  out
}

# Decrypts code according to curFunc
decrypt <- function(code, testword)
{
  out <- code
  index <- 1
  # for each character in the message, decode it according to the testword
  for (i in 1:nchar(message))
  {
    charInd <- charIndex(substr(code,i,i))
    codewordInd <- charIndex(substr(testword,index,index))
    if (charInd < 27)
    {
      # change the ith character to the character determined by the testword
      substr(out,i,i) <- rawToChar(as.raw(((charInd - codewordInd) %% 26 + 97)))
      index <- index + 1
      if (index > nchar(testword)){
        index = 1
      }
    }
  }
  out
}

# codemess holds the scrambled message
codemess <- encrypt(message, codeword)

# instantiate a map to hold previously computed codes' scores
map <- new.env(hash=T, parent=emptyenv())

# we begin with a basic (a->a, z->z) function for decrypting the codemess
curFunc <- strrep("a",3)

# calculate the score for curFunc and store it in the map
oldScore <- score(decrypt(codemess,curFunc))

bestLetter1 <- 1
bestLetter2 <- 1
bestLetter3 <- 1
```

```
start_time <- Sys.time()
for(test1 in 1:26){
  for(test2 in 1:26){
    for(test3 in 1:26){
      substr(curFunc,1,1) <- rawToChar(as.raw(test1 + 96))
      substr(curFunc,2,2) <- rawToChar(as.raw(test2 + 96))
      substr(curFunc,3,3) <- rawToChar(as.raw(test3 + 96))

      newScore <- score (decrypt(codemess,curFunc))
      if (newScore > oldScore){
        oldScore <- newScore
        bestLetter1 <- test1
        bestLetter2 <- test2
        bestLetter3 <- test3
      }
    }
  }
}

substr(curFunc,1,1) <- rawToChar(as.raw(bestLetter1 + 96))
substr(curFunc,2,2) <- rawToChar(as.raw(bestLetter2 + 96))
substr(curFunc,3,3) <- rawToChar(as.raw(bestLetter3 + 96))

if(decrypt(codemess,curFunc) == message)
{
  print(Sys.time() - start_time)
}
```

C. Optimized Brute Force Code (Known)

```
# we begin with a basic (a->a, z->z) function for decrypting the codemess
curFunc <- strrep("a",14)

start_time <- Sys.time()
for(i in 1:(nchar(curFunc))){
  bestLetter <- 1
  oldScore <- score(decrypt(codemess,curFunc))
  for(test in 1:26){
    substr(curFunc,i,i) <- rawToChar(as.raw(test + 96))
    newScore <- score (decrypt(codemess,curFunc))
    if (newScore > oldScore){
      oldScore <- newScore
      bestLetter <- test
    }
  }
  substr(curFunc,i,i) <- rawToChar(as.raw(bestLetter + 96))
}

if(decrypt(codemess,curFunc) == message)
{
  print(Sys.time() - start_time)
}
```

D. MCMC Code (Known)

```
# we begin with a basic (a->a, z->z) function for decrypting the codemess
curFunc <- strrep("a",nchar(codeword))

# calculate the score for curFunc and store it in the map
```

```
oldScore <- score(decrypt(codemess,curFunc))
map[[paste(curFunc, collapse='')]] <- oldScore

start_time <- Sys.time()
for(iteration in 1:2000){
  # sample two letters to swap
  oldFunc <- curFunc
  swap <- sample(1:26,1)
  index <- sample(1:nchar(codeword), 1)

  substr(curFunc,index,index) <- rawToChar(as.raw(swap + 96))
  newScore <- score (decrypt(codemess,curFunc))

  # if we have already scored this decoding,
  # retrieve score from our map
  if (exists(paste(curFunc, collapse =''), map)){
    newScore <- map[[paste(curFunc, collapse ='')]]
  } else
    # if we have not already scored this decoding,
    # calculate it and store it in the map
  {
    newScore <- score (decrypt(codemess,curFunc))
    map[[paste(curFunc, collapse = '')]] <- newScore
  }

  # decide whether to accept curFunc or to revert to oldFunc
  if (runif(1) > exp(newScore-oldScore))
  {
    curFunc <- oldFunc
  } else
  {
    oldScore <- newScore
  }

  # print out time elapsed
  if(decrypt(codemess,curFunc) == message)
  {
    print(Sys.time() - start_time)
    print(c(iteration,decrypt(codemess,curFunc)))
    print(curFunc)
    break
  }
}
```

E. Optimized Brute Force Code (Unknown)

```
# calculate the score for curFunc and store it in the map
start_time <- Sys.time()
bestWord <- "a"
bestScore <- oldScore
for(i in 1:(nchar(message))){
  curFunc <- strrep("a",i)
  oldScore <- score(decrypt(codemess,curFunc))
  for(i in 1:(nchar(curFunc))){
    bestLetter <- 1
    oldScore <- score(decrypt(codemess,curFunc))
    for(test in 1:26){
      substr(curFunc,i,i) <- rawToChar(as.raw(test + 96))
      newScore <- score(decrypt(codemess,curFunc))
      if (newScore > oldScore){
        oldScore <- newScore
        bestLetter <- test
      }
    }
    substr(curFunc,i,i) <- rawToChar(as.raw(bestLetter + 96))
  }
  if(bestScore*(1.1) < oldScore){
    print(c(i,decrypt(codemess,bestWord)))
    print(Sys.time() - start_time)
    print(bestWord)
    print(bestScore)
    break
  } else if(bestScore < oldScore){
    bestScore <- oldScore
    bestWord <- curFunc
  }
}
```

F. Optimized MCMC (Unknown)

```
# calculate the score for curFunc and store it in the map
curFunc <- "a"
oldScore <- score(decrypt(codemess,curFunc))
bestWord <- "a"
bestScore <- oldScore
start_time <- Sys.time()
for(i in 1:(nchar(message))){
  map <- new.env(hash=T, parent=emptyenv())
  curFunc <- strrep("a",i)
  oldScore <- score(decrypt(codemess,curFunc))
  for(iteration in 1:(150*i)){
    # sample two letters to swap
    oldFunc <- curFunc
    swap <- sample(1:26,1)
    index <- sample(1:nchar(curFunc), 1)
    substr(curFunc,index,index) <- rawToChar(as.raw(swap + 96))

    # if we have already scored this decoding,
    # retrieve score from our map
    if (exists(paste(curFunc, collapse =''), map)){
      newScore <- map[[paste(curFunc, collapse ='')]]
    } else
    {
      newScore <- score (decrypt(codemess,curFunc))
      map[[paste(curFunc, collapse = '')]] <- newScore
    }

    # decide whether to accept curFunc or to revert to oldFunc
    if (runif(1) > exp(newScore-oldScore))
    {
      curFunc <- oldFunc
    } else
    {
      oldScore <- newScore
    }
  }
  if(bestScore*(1.1) < oldScore){
    print(c(i,decrypt(codemess,curFunc)))
    print(Sys.time() - start_time)
    print(curFunc)
    print(oldScore)
    break
  } else if(bestScore < oldScore){
    bestScore <- oldScore
    bestWord <- curFunc
  }
  print(c(i, bestScore, oldScore))
}
```

G. Optimized Kasiski MCMC (Unknown)

```
# calculate factors
start_time <- Sys.time()
factorList <- c()
index <- 1
threeLetterMap <- new.env(hash=T, parent=emptyenv())
codemessNoSpace <- gsub(" ", "", codemess, fixed = TRUE)

# find repeated strings of length 3
for (i in 1:nchar(codemessNoSpace))
{
  if(i < nchar(codemessNoSpace)-1){
    seq <- substr(codemessNoSpace,i,i+2)
    if (exists(paste(seq, collapse =''), threeLetterMap)){
      seen <- threeLetterMap[[paste(seq, collapse ='')]]
      threeLetterMap[[paste(seq, collapse = '')]] <- seen+1
    } else
    {
      threeLetterMap[[paste(seq, collapse = '')]] <- 1
    }
  }
}

for (item in ls(threeLetterMap)) {
  if(threeLetterMap[[item]] > 1){
    factorList[index] <- item
    index <- index + 1
  }
}

# find the indexes of repeated strings of length 3
threeLetterMap <- new.env(hash=T, parent=emptyenv())
for (i in 1:nchar(codemessNoSpace))
{
  if(i < nchar(codemessNoSpace)-1){
    seq <- substr(codemessNoSpace,i,i+2)
    if (seq %in% factorList){
      if (exists(paste(seq, collapse =''), threeLetterMap)){
        list <- threeLetterMap[[paste(seq, collapse ='')]]
        threeLetterMap[[paste(seq, collapse = '')]] <- c(list, i)
      } else
      {
        threeLetterMap[[paste(seq, collapse = '')]] <- c(i)
      }
    }
  }
}
```

```r
# find the difference in indices
factorList <- c()
index <- 1
for (item in ls(threeLetterMap)) {
  previousElement <- 0
  for(element in threeLetterMap[[item]]){
    if(previousElement != 0){
      factorList[index] <- element - previousElement
      index <- index + 1
    }
    previousElement <- element
  }
}

# Factor an integer x; code for this function only is obtained from link below
# https://stackoverflow.com/questions/6424856/r-function-for-returning-all-factors
factor <- function(x) {
  x <- as.integer(x)
  div <- seq_len(abs(x))
  factors <- div[x %% div == 0L]
  factors <- list(neg = -factors, pos = factors)
  return(factors)
}

# find the intersection of their factors
factors <- c()
for (item in factorList) {
  if(length(factors) != 0){
    factors <- c(factors, factor(item)$pos)
  } else {
    factors <- factor(item)$pos
  }
}
numberOfOnes <- sort(table(factors),decreasing=TRUE)[2]
factors <- as.numeric(names(which(sort(table(factors),decreasing=TRUE) > numberOfOnes/3)))

# print time to process factors
print(Sys.time() - start_time)

# calculate the score for curFunc and store it in the map
curFunc <- "a"
oldScore <- score(decrypt(codemess,curFunc))
bestWord <- "a"
bestScore <- oldScore

for(i in sort(factors, decreasing=FALSE)){
  map <- new.env(hash=T, parent=emptyenv())
  curFunc <- strrep("a",i)
  oldScore <- score(decrypt(codemess,curFunc))
  for(iteration in 1:(150*i)){
    # sample two letters to swap
    oldFunc <- curFunc
    swap <- sample(1:26,1)
    index <- sample(1:nchar(curFunc), 1)
    substr(curFunc,index,index) <- rawToChar(as.raw(swap + 96))

    # if we have already scored this decoding,
    # retrieve score from our map
    if (exists(paste(curFunc, collapse =''), map)){
      newScore <- map[[paste(curFunc, collapse ='')]]
    } else
    {
      newScore <- score (decrypt(codemess,curFunc))
      map[[paste(curFunc, collapse = '')]] <- newScore
    }

    # decide whether to accept curFunc or to revert to oldFunc
    if (runif(1) > exp(newScore-oldScore))
    {
      curFunc <- oldFunc
    } else
    {
      oldScore <- newScore
    }
  }
  if(bestScore*(1.1) < oldScore){
    print(c(i,decrypt(codemess,curFunc)))
    print(Sys.time() - start_time)
    print(curFunc)
    print(oldScore)
    break
  } else if(bestScore < oldScore){
    bestScore <- oldScore
    bestWord <- curFunc
  }
}
```

H. Trial Data

Note: All values are in seconds.

Brute Force, no optimization

Word Length	1	2
Trial 1	0.6116588	16.67368
Trial 2	0.620575	19.58916
Trial 3	0.6274989	18.97162
Trial 4	0.6199031	17.77639
Trial 5	0.6971471	17.48337
Mean	**0.6354**	**18.0988**
St Dev	**0.0350**	**1.1722**

Known code length decryption

Code Word	"Lemon"				"Lemonextratext"			
Algorithm	Brute Force	Matching	Iterations	No Map	Brute Force	Matching	Iterations	No Map
Trial 1	3.118933	7.992589	5.77893	11.42185	9.52847	25.42678	23.59009	48.18826
Trial 2	3.112635	4.424063	6.170928	12.35393	9.215091	41.51789	29.82598	19.85455
Trial 3	3.027324	8.462491	7.816761	10.4657	9.205712	28.21652	40.99496	39.04337
Trial 4	3.047137	12.34381	9.178447	10.75784	9.19666	43.8243	34.25411	16.14839
Trial 5	3.056956	6.177106	9.318182	8.415107	10.02758	33.90807	29.42386	30.90111
Mean	**3.0726**	**7.8800**	**7.6526**	**10.6829**	**9.4347**	**34.5787**	**31.6178**	**30.8271**
St Dev	**0.0409**	**2.9631**	**1.6459**	**1.4607**	**0.3597**	**8.0361**	**6.4682**	**13.2739**

Unknown code length decryption

Codelength	5		14		50	
Algorithm	Brute Force	Iterations	Brute Force	Iterations	Brute Force	Iterations
Trial 1	10.02168	23.23882	78.18348	229.27812	872.8584	N/A
Trial 2	10.50066	20.81306	75.20466	194.57232	846.963	N/A
Trial 3	10.76511	17.76072	77.24238	236.0265	875.46	N/A
Trial 4	10.10328	20.20331	79.56582	226.72278	908.667	N/A
Trial 5	10.06669	18.91866	79.16832	221.90706	854.0712	N/A
Mean	**10.2915**	**20.1869**	**77.8729**	**221.7014**	**875.9871**	**N/A**
St Dev	**0.3267**	**2.0737**	**1.7434**	**15.9989**	**25.3011**	**N/A**