

# Lab\_04

March 3, 2025

Probability for Data Science

UC Berkeley, Spring 2025

Michael Xiao and Ani Adhikari

CC BY-NC-SA 4.0

This content is protected and may not be shared, uploaded, or distributed.

```
[10]: from datascience import *
      from prob140 import *
      import numpy as np
      from scipy import special
      import pickle
```

```
[11]: # Run this code

import itertools as it
from collections import Counter

def clean_string(string):
    """
    Cleans an input string by replacing all newlines with spaces, and then
    removing all letters not in *allowable_letters*
    """
    string = string.replace("\n", " ")
    return "".join([i for i in string.lower().strip() if i in
↪allowable_letters])

def load_bigrams(text):
    """
    Takes a string which has already been cleaned, and returns a dictionary
    of conditional bigram probabilities

     $cond\_bigram[(a,b)] = P(X_{n+1} = b \mid X_n = a)$ 

    Uses Laplace smoothing with size $1$, to remove zero transition
↪probabilities
    """
```

```

bigram_counter = Counter(list(it.product(allowable_letters, repeat=2)))
gram_counter = Counter(allowable_letters*len(allowable_letters))
for l1, l2 in zip(text, text[1:]):
    bigram_counter[(l1, l2)] += 1
    gram_counter[l1] += 1
cond_bigram = {k:v/gram_counter[k[0]] for k,v in bigram_counter.items()}
return cond_bigram

def bigram_from_file(filename):
    """
    Given a filename, this reads it, cleans it, and returns the conditional_
    ↪bigram
    """
    file_text = open(filename).read()
    file_text = clean_string(file_text)
    return load_bigrams(file_text)

def reverse_cipher(cipher):
    return {v:k for k, v in cipher.items()}

def print_differences(cipher1, cipher2):
    for k in cipher1:
        if cipher1[k] != cipher2[k]:
            print("%s: %s %s"%(k, cipher1[k], cipher2[k]))

def num_errors(cipher, encoded_text, original_text):
    decoded = np.array(list(decode_text(encoded_text, cipher)))
    original = np.array(list(original_text))
    num_errors = np.count_nonzero(decoded != original)
    return num_errors

def get_secret_text(student_id):
    with open('Lab_04_data/secret_strings.p', 'rb') as f:
        texts = pickle.load(f)
    return texts[student_id % len(texts)]

```

# 1 Lab 4: Code Breaking by MCMC (Due Monday, March 3rd at 5 PM)

## 1.1 Instructions

Similar to your homeworks, your labs will generally have two components: a written portion and a portion that also involves code.

- Written work should be completed on paper, and coding questions should be done in the notebook.
- Start the work for the written portions of each section on a new page. - You are welcome to  $\text{\LaTeX}$  your answers to the written portions, but staff will not be able to assist you with  $\text{\LaTeX}$  related issues.
- Show your work. Give reasoning. The question isn't always going to ask for it, because

we assume that you will provide justification for your answers. Every answer should contain a calculation, reasoning, or diagrams that are clearly labeled to show what's going on. - It is your responsibility to ensure that both components of the lab are submitted completely and properly to Gradescope. **Make sure to assign each page of your pdf to the correct question. - Refer to the bottom of the notebook for submission instructions.**

## 1.2 Identify Your Lab Partner

This is a multiple choice question. Please select **ONE** of following options that best describes how you complete this lab.

- I am doing this lab by myself and I don't have a partner.
- My partner for this lab is [PARTNER'S NAME] with email [berkeley.edu email address]. [SUBMITTER'S NAME] will submit to Gradescope and add the other partner to the group on Gradescope after submission.

Please copy and paste **ONE** of above statements and fill in blanks if needed. If you work with a partner, make sure only one of you submit on Gradescope and that the other member of the group is added to the submission on Gradescope. Refer to the bottom of the notebook for submission instructions.

I am doing this lab by myself.

### 1.3 Breaking a Substitution Code

Cryptography is the study of algorithms used to encode and decode messages. Markov Chain Monte Carlo (MCMC) methods have been successfully used to decode messages encrypted using substitution codes and also [more complex](#) encryption methods. In this lab you will apply MCMC and the Metropolis algorithm to decode English text that has been encrypted with a substitution code.

The lab is based on the paper [The Markov Chain Monte Carlo Revolution](#) by [Persi Diaconis](#). It was presented at the 25th anniversary celebrations of MSRI up the hill, and appeared in the Bulletin of the American Mathematical Society in November 2008. The code is based on [Simulation and Solving Substitution Codes](#) written by [Stephen Connor](#) in 2003.

Recall that in class we worked with a tiny alphabet and a short message that had been encoded using a substitution code. We used a Markov chain model and a scoring system to choose the decoder that had the highest score. That is, we chose the decoder that had the highest likelihood given the message.

Because the alphabet was small, we were able to list all the decoders and their corresponding scores. When the alphabet is large, making an exhaustive list of all possible decoders and scores is not feasible. That's where the Monte Carlo part of the algorithm comes in. The idea is to choose decoders at random using an algorithm that favors good decoders over bad ones.

In this lab you will work with an alphabet consisting of all 26 English letters as well as a space character that will have a special status. **Before you begin, please review Sections 11.2 (Code Breaking) and 11.3 (Metropolis Algorithm) of the textbook.** The lab follows those sections closely.

Unlike all the other labs, this lab is not divided into parts. It's just one small project. You will learn how to: - Apply a decoder to encoded text - Use transformations for improved numerical accuracy - Implement the Metropolis algorithm - Decode a message encrypted by a substitution code

The programming needed for this lab is more complex than what can reasonably be expected based on just Data 8 background, so much of it has been done for you. Those of you who have more extensive Python knowledge might be interested in looking at the details.

Please start by running all the cells above this one.

### 1.4 Markov Chain on Characters: Alphabet and Bigrams

The text that you are going to decode was written in English. For data on the frequencies of all the different bigrams in English, we will start by counting all the bigrams in *War and Peace* by Tolstoy. That is one of the longest novels in English (actually in Russian, but we are using an English translation from Project Gutenberg) and is often used as a “corpus” or body of language on which to base analyses of other text.

To keep the calculations manageable, we will restrict ourselves to a 27-character alphabet, consisting of the 26 lower case letters and a space. The cell below places all 27 in a list.

```
[12]: allowable_letters = list("abcdefghijklmnopqrstuvwxyz ")
```

In the cell below, we find the relative frequencies of all bigrams in War and Peace and place them in the dictionary `wp_bigrams`. You don't need to know what a dictionary is. It's fine to imagine it containing the same information as a Table that has a column with the bigrams and another column with the relative frequencies.

```
[13]: wp_bigrams = bigram_from_file("Lab_04_data/warandpeace.txt")
```

### 1.4.1 1. Transition Matrix

As in class, we model the sequence of characters in the text as a Markov chain. This model seems overly simplified, but it turns out to be enough for many purposes such as the one in this lab.

The state space of this chain is the alphabet of 27 characters. Of course the text has other punctuation, but we are stripping all of it for simplicity. We are also replacing upper case letters with lower case ones.

Construct the transition matrix `wp_bigram_mc` for this Markov chain based on the relative frequencies in `wp_bigrams` defined above. Refer to [Section 10.1.5](#) for an example that uses relevant code.

We have started the code off for you by defining the transition function `wp_transition`. Use `allowable_letters` in your definition.

```
[14]: def wp_transition(a, b):
        if (a, b) in wp_bigrams:
            return wp_bigrams[(a, b)]
        return 0

wp_bigram_mc = MarkovChain.from_transition_function(allowable_letters,
↪wp_transition)
wp_bigram_mc
```

```
[14]:
```

	a	b	c	d	e	f	g	\
a	0.000123	0.016969	0.034439	0.055522	0.000809	0.008217	0.016857	
b	0.092429	0.006631	0.000144	0.000490	0.327913	0.000029	0.000029	
c	0.109112	0.000016	0.017047	0.000162	0.213245	0.000016	0.000016	
d	0.021188	0.000659	0.000110	0.011046	0.116413	0.000963	0.003702	
e	0.042608	0.000884	0.017399	0.091233	0.025779	0.010148	0.006876	
f	0.070639	0.000564	0.000692	0.000419	0.083948	0.051705	0.000036	
g	0.066342	0.000058	0.000117	0.001013	0.114711	0.000234	0.008724	
h	0.164932	0.000412	0.000388	0.000382	0.448388	0.000424	0.000024	
i	0.016801	0.007746	0.049227	0.050081	0.048754	0.021611	0.024772	
j	0.038432	0.000384	0.000384	0.000384	0.217525	0.000384	0.000384	
k	0.027715	0.000635	0.002493	0.000196	0.277055	0.000880	0.000831	
l	0.079685	0.000860	0.001119	0.071669	0.175475	0.023976	0.000963	
m	0.151252	0.017642	0.000616	0.000081	0.245638	0.002254	0.000032	
n	0.030173	0.000858	0.050129	0.187753	0.080584	0.004978	0.135691	
o	0.007304	0.004705	0.007650	0.016043	0.002656	0.088668	0.004055	
p	0.100020	0.000285	0.002809	0.000066	0.180110	0.002217	0.000241	

q	0.000424	0.000424	0.000424	0.000424	0.000424	0.000424	0.000424
r	0.055996	0.001495	0.009956	0.029470	0.240263	0.003058	0.010030
s	0.051883	0.002719	0.013724	0.000546	0.107847	0.001921	0.000307
t	0.034031	0.000150	0.002893	0.000022	0.084535	0.000839	0.000066
u	0.020441	0.014008	0.031733	0.022107	0.029450	0.005122	0.050631
v	0.066239	0.000074	0.000037	0.000406	0.535332	0.000037	0.000148
w	0.205470	0.000203	0.000405	0.006229	0.130987	0.000844	0.000135
x	0.076627	0.000227	0.136930	0.000227	0.071639	0.000453	0.000227
y	0.028471	0.003413	0.003737	0.000410	0.056121	0.003305	0.000281
z	0.057167	0.002071	0.000829	0.011599	0.304060	0.000414	0.000414
	0.119834	0.043355	0.036531	0.029719	0.020259	0.036824	0.016908

	h	i	j	...	r	s	t	u	\
a	0.001598	0.042852	0.000829	...	0.087455	0.097825	0.138167	0.012351	
b	0.000086	0.029493	0.005074	...	0.057228	0.013925	0.007496	0.152223	
c	0.188266	0.041799	0.000016	...	0.033608	0.001622	0.067070	0.027444	
d	0.000676	0.066075	0.002003	...	0.030282	0.020191	0.000625	0.009432	
e	0.002347	0.011143	0.000273	...	0.140364	0.065150	0.023902	0.001018	
f	0.000182	0.085168	0.000036	...	0.103246	0.004060	0.039234	0.033918	
g	0.116172	0.048798	0.000019	...	0.053451	0.017486	0.004595	0.026112	
h	0.000084	0.155101	0.000006	...	0.007752	0.001135	0.025150	0.009562	
i	0.000040	0.002659	0.000006	...	0.034191	0.125124	0.122580	0.000756	
j	0.000384	0.004612	0.000384	...	0.000384	0.000384	0.000384	0.455419	
k	0.040375	0.166390	0.000049	...	0.004839	0.037785	0.000440	0.033092	
l	0.000166	0.098503	0.000010	...	0.004878	0.016964	0.017234	0.015773	
m	0.000292	0.080117	0.000016	...	0.003211	0.031359	0.003421	0.027127	
n	0.000847	0.033772	0.000657	...	0.000429	0.036715	0.093331	0.005266	
o	0.001918	0.011763	0.000639	...	0.103606	0.035482	0.052531	0.123395	
p	0.009196	0.092294	0.000022	...	0.172428	0.023770	0.053686	0.025636	
q	0.000424	0.000424	0.000424	...	0.000424	0.000424	0.000424	0.988545	
r	0.001536	0.089210	0.000027	...	0.033242	0.060522	0.031234	0.020437	
s	0.066675	0.055946	0.000031	...	0.000184	0.059598	0.109596	0.024281	
t	0.325798	0.061567	0.000018	...	0.025040	0.026859	0.019590	0.013664	
u	0.000740	0.027151	0.000031	...	0.125189	0.137731	0.169911	0.000046	
v	0.000074	0.176772	0.000037	...	0.006712	0.022719	0.002877	0.000848	
w	0.201351	0.169258	0.000017	...	0.008947	0.014603	0.000405	0.000523	
x	0.013829	0.116527	0.000227	...	0.000227	0.001587	0.107005	0.004081	
y	0.000648	0.024453	0.000043	...	0.002463	0.033893	0.018361	0.001318	
z	0.064209	0.101906	0.000414	...	0.000414	0.001243	0.000829	0.012842	
	0.085827	0.053078	0.002858	...	0.026106	0.070295	0.150823	0.009706	

	v	w	x	y	z	
a	0.021411	0.010605	0.000255	0.025883	0.001893	0.074040
b	0.001355	0.000605	0.000029	0.073546	0.000029	0.005824
c	0.000016	0.000162	0.000016	0.005044	0.000097	0.009878
d	0.002721	0.000549	0.000008	0.010066	0.000025	0.641109
e	0.017494	0.010275	0.009970	0.012008	0.000604	0.353170

f	0.000018	0.000601	0.000018	0.001566	0.000018	0.353724
g	0.000019	0.000526	0.000019	0.001908	0.000136	0.431058
h	0.000036	0.000257	0.000006	0.006134	0.000006	0.092114
i	0.020112	0.000046	0.001874	0.000012	0.003426	0.031365
j	0.000384	0.000384	0.000384	0.000384	0.000384	0.000384
k	0.000684	0.004399	0.000049	0.006648	0.000049	0.247287
l	0.004578	0.004961	0.000010	0.103671	0.000269	0.133903
m	0.000114	0.000259	0.000016	0.039856	0.000016	0.202474
n	0.004093	0.000727	0.000451	0.011362	0.000125	0.220043
o	0.030939	0.052007	0.000744	0.003762	0.000629	0.140942
p	0.003995	0.000527	0.000022	0.007287	0.000022	0.069138
q	0.000424	0.000424	0.000424	0.000424	0.000424	0.000849
r	0.004756	0.002054	0.000007	0.038738	0.000559	0.212013
s	0.000491	0.004389	0.000006	0.002320	0.000043	0.385687
t	0.004346	0.005251	0.000009	0.013898	0.002946	0.257271
u	0.001913	0.000046	0.000679	0.000339	0.001296	0.059255
v	0.000111	0.000295	0.000037	0.004057	0.000037	0.098473
w	0.000017	0.000456	0.000017	0.000709	0.000017	0.139512
x	0.035593	0.000227	0.036046	0.001587	0.000227	0.084335
y	0.000562	0.002139	0.000022	0.000238	0.000238	0.678180
z	0.001657	0.000829	0.000414	0.006214	0.019884	0.054681
	0.006878	0.070007	0.000777	0.011618	0.000255	0.025147

[27 rows x 27 columns]

## 1.5 Decoders

As you know, when a substitution code has been used for encryption, a decoder is a permutation of the alphabet. We apply this permutation to our encoded text in order to generate the decoded text.

We will be representing the decoder as a Python dictionary. Here is an example of how a dictionary works, using a three-letter alphabet 'a', 'b', 'c'.

If the decoder is ['b', 'c', 'a'] then the substitutions for decoding the encrypted message are

$$a \rightarrow b \quad b \rightarrow c \quad c \rightarrow a$$

This decoder written as a dictionary looks like this:

```
simple_decoder = {'a':'b','b':'c','c':'a'}
```

To access a value in the decoder, use the method `.get`. For example, to access the translation of *a*, use

```
simple_decoder.get('a')
```

```
[15]: simple_decoder = {'a':'b', 'b':'c', 'c':'a'}
      simple_decoder.get('a')
```

```
[15]: 'b'
```

## 1.6 Keeping Spaces Intact

To simplify calculations, we are assuming that our substitution code keeps spaces unchanged. Letters can precede and follow spaces, but spaces will be fixed points in the encoding permutation.

So our decoder must keep spaces unchanged as well. The following cell defines a function called `random_decoder` which will randomly generate a decoder. It will operate on the 26 letters, omitting the space.

Run the cell below. The function `random_decoder` constructs a random permutation of the alphabet and places it in a dictionary to make it a decoder.

```
[16]: # We don't operate on spaces
decoder_letters = np.array(list("abcdefghijklmnopqrstuvwxyz"))

# Set up a dictionary with one entry for each letter
identity_decoder = {letter:letter for letter in decoder_letters}

# Random starting decoder
def random_decoder():
    """ Random decoder """
    new_letters = decoder_letters.copy()
    np.random.shuffle(new_letters)
    return {orig:new for orig ,new in zip(decoder_letters, new_letters)}
```



### 1.6.1 2. Applying a Decoder

Define a function `decode_text` that takes as arguments a `string` to be decoded and the `decoder`. It should return the decoded string.

Remember that we are only decoding alphabetical characters. If a character is not in the decoder (e.g. a space), leave the character alone.

```
[17]: def decode_text(string, decoder):
    new_string = ""

    for char in string:
        if char in decoder:
            new_letter = decoder.get(char)
        else:
            new_letter = char

        # Now we append the letter to the back of the new string
        new_string = new_string + new_letter

    return new_string
```

Run the cell below to check that your function works as expected.

```
[18]: simple_decoder = {'a':'b', 'b':'c', 'c':'a'}
    decode_text("abcba", simple_decoder)
```

```
[18]: 'bcacb'
```

## 1.7 Score of a Decoder

For letters  $x_1$  and  $x_2$ , let  $B(x_1, x_2)$  be the  $(x_1, x_2)$ th entry in the bigram transition matrix. Suppose the decoded text consists of the characters  $x_0, x_1, \dots, x_n$ . Recall from class that we defined the *score* of the corresponding decoder  $i$  to be

$$s(i) = \prod_{k=0}^{n-1} B_i(x_k, x_{k+1})$$

Here's a tweak to what was done in class. When the string is long, the score of any decoder is going to be very small. This causes problems with numerical accuracy. So we will work with the log-score instead, that is, with the log of the score.

$$\log(s(i)) = \sum_{k=0}^{n-1} \log(B_i(x_k, x_{k+1}))$$

### 1.7.1 3. Log Probability of Path

Define a function `log_prob_path` that takes two arguments: - a string `message` that is a sequence of letters - the bigram transition matrix `mc` of a `MarkovChain`

and returns the log of the probability of the path `message` taken by `mc`, given that the first character of `message` is the initial state of the path.

Though `mc.prob_of_path` returns the probability of the path, don't use `np.log(mc.prob_of_path)`. Each path has very small probability, so the numbers will round too soon and your computations might be inaccurate.

Instead, you can use the function `mc.log_prob_of_path`. This takes as its first argument the starting character of the string, and as its next argument a list or array containing the characters in the rest of the string. It returns the log of the probability of the path, conditional on the starting character.

Some string methods will be helpful:

- `string[0]` returns the first letter of a string. For example:

```
>>> x = 'HELLO'
>>> x[0]
'H'
```

- `string[1:]` returns everything except the first letter of a string. For example:

```
>>> x[1:]
'ELLO'
```

- `list(string)` splits the string into a list of its characters. This was used above to generate `allowable_letters`.

```
[19]: def log_prob_path(string, bigram_mc):
      start = string[0]

      path = list(string[1:])

      return bigram_mc.log_prob_of_path(start, path)
```

### 1.7.2 4. Log Score of Decoder

Define a function `log_score` that takes the following arguments:

- an encrypted string `string`
- a decoder `decoder`
- a bigram transition matrix `bigrams`

The function should decode the encrypted string using the decoder, and return the log score of the decoder.

Use the functions `decode_text` and `log_prob_path` that you wrote earlier.

```
[20]: def log_score(string, decoder, bigrams):  
  
    decoded = decode_text(string, decoder)  
    return log_prob_path(decoded, bigrams)
```

## 1.8 Markov Transitions on Decoders: A Proposal Matrix

The Metropolis algorithm we will use is essentially the same as the one defined in class and proved to converge to the correct distribution. There are tweaks to increase computational accuracy and to deal with spaces, but you shouldn't worry about either of them.

Please have Section 11.3 of the textbook at hand as you go through the exercises below.

Our goal is to find the “best” decoder, that is, the decoder that has the highest score. Since we can't list all the decoders, we are going to try out decoders at random – but not uniformly at random. So we need a rule for moving from one decoder to another.

The Metropolis algorithm starts by creating an essentially arbitrary transition matrix on decoders. Given that we have just tried out decoder  $i$ , we will pick another decoder to try out, using a probability distribution that just satisfies a couple of simple conditions. Think of this as a possibly dumb starting point. We'll use it as a base to create a smarter process.

You will now define the starting point, which is a transition matrix  $Q$  on the space of all decoders, satisfying some conditions needed for the Metropolis algorithm. The purpose of  $Q$  is to propose a decoder  $j$  for us to try out next, given that we have just tried out decoder  $i$ . We will still have to decide whether or not to accept that proposal, but we'll get to that later. At this stage, we will simply define  $Q$ .

- Define  $Q$  as follows: given that the chain is at decoder  $i$ , pick another decoder by randomly swapping two elements of  $i$ . For any two decoders  $i$  and  $j$ , let

$$Q(i, j) = \begin{cases} \frac{1}{\binom{26}{2}} & \text{if } i \text{ and } j \text{ differ in exactly two places} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Two decoders differ in exactly two places when two letters have “switched” positions. For example, in a 3 letter alphabet the decoders `abc` and `bac` differ in exactly two places because `a` and `b` have “switched” positions.

You will show below that  $Q$  satisfies the conditions for the proposal transition matrix used in the Metropolis algorithm.

### 1.8.1 5a) Properties of $Q$

$Q$  is a huge matrix. How many rows does it have? For reference, `special.comb(n, k)` evaluates to  $\binom{n}{k}$  and `special.factorial(n, exact=True)` evaluates to  $n!$ . The modules were imported at the start of the lab.

```
[21]: special.factorial(26, exact=True)
```

```
[21]: 403291461126605635584000000
```

Define the three decoders (that is, permutations) below:

- `decoder_1`: alphabetical order, that is, `abcdef ... xyz`
- `decoder_2`: `bacdef ... xyz`, that is, `ba` followed by the other letters in alphabetical order
- `decoder_3`: `badcef ... xyz`, that is, `badc` followed by the other letters in alphabetical order

For each of (i)-(iv) below, type  $1/(26\_choose\_2)$  if you think the value is  $1/\binom{26}{2}$  and 0 otherwise, and enter the reason for your choice.

(i)  $Q(\text{decoder\_1}, \text{decoder\_2}) = 1/(26\_choose\_2)$

because they differ in exactly 2 positions (a,b), but all other letters follow the same order.

(ii)  $Q(\text{decoder\_2}, \text{decoder\_1}) = 1/(26\_choose\_2)$

because they differ in exactly 2 positions (a,b). Since the transition matrix  $Q$  is symmetric ( $Q(i,j) = Q(j,i)$ ), the probability remains the same even though the order of decoders is reversed.

(iii)  $Q(\text{decoder\_1}, \text{decoder\_3}) = 0$

because they differ in exactly 4 positions (a,b,c,d). The letters are too scrambled and require more than a single swap of two letters to be decoded.

(iv)  $Q(\text{decoder\_2}, \text{decoder\_3}) = 1/(26\_choose\_2)$

because they differ in exactly 2 positions (c,d), but all other letters follow the same order.

**Symmetric Transition Matrix:** Explain why  $Q$  is a symmetric transition matrix.

[Note: You have to establish that  $Q$  is symmetric and that it is a transition matrix. For the latter, it is a good idea to focus on the row corresponding to `decoder_1` above. What criteria do the entries of each row have to satisfy? Why are the criteria satisfied?]

$Q$  is a symmetric transition matrix because for any two states  $i$  and  $j$ , the transition probability satisfies  $Q(i,j) = Q(j,i)$ . Symmetry is established because the transition probability depends only on whether the states  $i$  and  $j$  differ in exactly two positions, and swapping those two positions is symmetric. The criteria for the entries of each row in  $Q$  are that they must be non-negative and sum to 1. The non-negative condition is satisfied because  $Q(i,j)$  is either  $1/(26\_choose\_2)$  or 0. The sum of each row must also equal 1, which is satisfied because for any state  $i$ , there are  $(26\_choose\_2)$  possible states  $j$  that differ from  $i$  in exactly two positions, and each transition probability is  $1/(26\_choose\_2)$ . Therefore,  $(26\_choose\_2) * 1/(26\_choose\_2) = 1$ . This uniform probability distribution, where any valid transition between two decoders has an equal chance of being chosen, ensures that  $Q$  is a symmetric transition matrix.

**Irreducible:** It is a fact that  $Q$  is the transition matrix of an irreducible chain, but formally establishing that takes a bit of writing. To avoid complicated writing, let's reduce the problem to

an alphabet of just three letters **a**, **b**, and **c** and transition behavior defined analogously on the six decoders, with positive transition probabilities each equal to  $1/(3\_choose\_2) = 1/3$  if the two permutations differ in exactly two places. **Why does this create an irreducible chain?**

This creates an irreducible chain because any decoder can be reached from any other decoder in a finite number of transitions. When looking at the letters a, b, and c, the six possible decoders are abc, bac, cab, acb, bca, and cba. It is possible to transition to another decoder by swapping exactly two positions, as all six decoders are interconnected. No matter which decoder you start with, you can eventually reach any other decoder by a sequence of swaps, making the chain irreducible. This same idea applies to Q and the 26 letters in the alphabet. Each decoder can be reached from any other through a series of transitions involving swaps of exactly two positions.

For any size of alphabet, there is a [famous algorithm](#) for creating all possible permutations by successively swapping two *adjacent* elements. It was known to bell-ringers in 17th-century England. Adjacent swaps are among the transitions allowed by the matrix Q, so the chain is irreducible.

**Proposal Chain:** Now explain why Q satisfies all the criteria for the transition matrix of a proposal chain. You have done all the work already; carefully read [Section 11.3.1](#) to see what is needed.

Q satisfies all the criteria for the transition matrix of a proposal chain because it is symmetric, irreducible, and meets the conditions for a valid transition matrix. The entries of Q are non-negative because the transition probability is either  $1/(26\_choose\_2)$  or 0, depending on whether decoders i and j differ in exactly two positions. The sum of the entries in each row of Q equals 1, since for each decoder i, there are exactly  $(26\_choose\_2)$  decoders j that differ from i by two positions, and the transition probability for each is  $1/(26\_choose\_2)$ , leading to a row sum of 1. In addition, Q is symmetric because the transition probability depends solely on whether two decoders differ by exactly two positions, and swapping two positions is a symmetric operation. Therefore,  $Q(i, j) = Q(j, i)$ . Finally, the transition probabilities are uniform, meaning every valid transition between decoders has an equal probability of  $1/(26\_choose\_2)$ . Since Q meets these conditions, it satisfies all the requirements for a proposal chain.

### 1.8.2 5b) Making a Proposal to Move

Define the function `generate_proposed_decoder` which takes an existing decoder and returns a new decoder as follows: it randomly picks two different letters of the alphabet and swaps the decoding for those letters. Make sure to use `decoder_letters` rather than `allowable_letters` because we want to keep the spaces as is.

For example, if we have the decoder `{'a':'b', 'b':'c', 'c':'a'}` and decide to swap a and b, our new decoder should be `{'a':'c', 'b':'b', 'c':'a'}`.

There are several different ways you can use Python to draw random samples. In this instance we recommend `np.random.choice`. It takes two arguments: - a list from which to sample uniformly at random with replacement - the number of times to sample

The optional argument `replace=False` leads to simple random sampling.

It returns an array consisting of the sampled elements.

```
[22]: def generate_proposed_decoder(decoder):  
      new_decoder = decoder.copy()
```

```

letters = np.random.choice(
    list(decoder.keys()), size = 2, replace=False)

letter1 = letters[0]
letter2 = letters[1]

new_value_of_letter1 = new_decoder[letter2]
new_value_of_letter2 = new_decoder[letter1]

# This code replaces the value of letter1 and letter2
# with new_value_of_letter1 and new_value_of_letter2
new_decoder[letter1] = new_value_of_letter1
new_decoder[letter2] = new_value_of_letter2
return new_decoder

```

## 1.9 Markov Chain on Decoders: The Metropolis Algorithm

Remember that we are looking for the decoder that has the highest score. So, given that we have just tried out one decoder, we would like to move on to not just any other random decoder but a *better* decoder. That is, we'd like a process that is likely to move us to a decoder with a higher score.

The proposal matrix  $Q$  won't do this, because  $Q$  doesn't involve the scores at all. The Metropolis algorithm uses  $Q$  *as well as the scores* to help us reach our destination. You will now implement the details.

Imagine a probability distribution  $\pi$  with entries proportional to the scores. This distribution can be thought of as consisting of the likelihood of each decoder given the text. Suppose we can create a Markov chain in which Decoder  $i$  appears among our randomly selected decoders with target probability  $\pi(i)$  in the long run. Then the decoder we are most likely to see in the long run is the one with the highest likelihood.

The trouble is that there are too many decoders for us to list, so we can't normalize all the scores to get a probability distribution. But we don't actually need the entire distribution  $\pi$ . We are just looking for the mode of  $\pi$ .

### 1.9.1 Acceptance Ratios

For this, we are going to use the fact that for any two decoders  $i$  and  $j$ , we can easily see which one has the higher score. One way to capture this information is to calculate the ratio

$$r(i, j) = \frac{\pi(j)}{\pi(i)} = \frac{s(j)}{s(i)}$$

**The Metropolis algorithm starts with the proposal transition matrix  $Q$  and uses these ratios to create a new chain that has our target  $\pi$  as its limit distribution.**

Recall that if the new chain is at  $i$ , then the decision about whether or not to accept a transition to a new state  $j$  depends on comparing the ratio  $r(i, j)$  to the value 1.

**Required Reading at this point:** [Section 11.3.2](#) which discusses how to use the ratio  $r$  to decide whether or not to accept a proposal made by  $Q$ .

**Log Ratio of Scores** Because we are working with log scores and not the scores themselves, every statement that we made in class about scores has to be rewritten in terms of log scores. For example, the ratio can be written as

$$\frac{s(j)}{s(i)} = e^{\log\left(\frac{s(j)}{s(i)}\right)} = e^{\log(s(j)) - \log(s(i))}$$

You will have to calculate the ratios, so keep in mind that `np.e**(x)` evaluates to  $e^x$ .

Write the condition  $r(i, j) < 1$  in terms of  $\log(s(i))$  and  $\log(s(j))$ . Justify your answer.

Answer on paper.

## 1.10 Implementing the Metropolis Algorithm



### 1.10.1 6. An Encoded Message

Enter your student id in the cell below to access your secret encoded message. By the end of the lab, you should be able to decode your string.

```
[23]: student_id = 3038472751
      secret_text = get_secret_text(student_id)
      secret_text
```

```
[23]: 'koh lxxphnntquc aurnq khnk xn u cjkxknkuth uhqyfxr ruaurxke khnk kouk
aqtqhnnxwhve thkn cyqh sxllxrjvk un xk rypkxpjhn koh chkhq aurnq khnk dxvv
fhthp xp nhrypsn vxph ja uk koh nkuqk koh qjppxpt nahhs nkuqkn nvydve fjk thkn
lunkhq huro cxpjkx ulkhq eyj ohuq koxn nxtpuv fhha u nxptvh vua noyjvs fh
rycavhkhx huro kxch eyj ohuq koxn nyjps sxpt qhchcfhq ky qjp xp u nkquxtok vxph
ups qjp un vypt un aynnxfvh koh nhryps kxch eyj luxv ky rycavhkh u vua fhlyqh
koh nyjps eyjq khnk xn ywhq koh khnk dxvv fhthp yp koh dyqs nkuqk yp eyjq cuqb
thk qhuse nkuqk nyjqrh koh lxxphnntquc aurnq khnk'
```

### 1.10.2 7. Implementation

Define the function `metropolis` which will take as its arguments:

- the encrypted text, as a string `string_to_decode`
- the transition matrix of bigrams `bigrams`
- the number of repetitions for running the Metropolis algorithm `reps`

The function should return the decoder that it arrives at after performing the Metropolis algorithm the specified number of times.

If the number of repetitions is large, you know from class that the distribution of this random decoder is close to the desired distribution  $\pi$  which is the likelihood distribution of the decoder given the data. Therefore you expect to end up with a decoder that has a high likelihood compared to the other decoders.

Between iterations, you should keep track of the following variables:

1. `best_decoder`: the decoder that has the highest log score
2. `best_score`: the log score associated with the `best_decoder`
3. `decoder`: the decoder that you are currently working with
4. `last_score`: the log score of the current decoder

In each iteration, you should do the following:

1. Generate a new decoder based on the current decoder; the “new” decoder might be the same as the current one.
2. Calculate the log score of your new decoder.
3. **Important:** Follow the Metropolis algorithm to decide whether or not to move to a new decoder.
4. If the new decoder’s log score is the best seen so far, update `best_score` and `best_decoder`

```
[24]: import time

def p_coin(p):
    """
    Flips a coin that comes up heads with probability p
    and returns 1 if Heads, 0 if Tails
    """
    return np.random.random() < p

def metropolis(string_to_decode, bigrams, reps):
    decoder = random_decoder() # Starting decoder

    best_decoder = decoder
    best_score = log_score(
        string_to_decode,
        best_decoder,
        bigrams
    )
```

```

last_score = best_score

for rep in np.arange(reps):

    # This will print out our progress
    if rep*10%reps == 0: # Repeat every 10%
        time.sleep(.01)
        decoded_text = decode_text(
            string_to_decode,
            best_decoder
       )[:40]
        print('Score: %.00f \t Guess: %s'%(best_score, decoded_text))

    #####
    # Your code starts here #
    #####
    proposed_decoder = generate_proposed_decoder(decoder)
    log_s_orig = last_score
    log_s_new = log_score(string_to_decode, proposed_decoder, bigrams)

    # If better than before or p-coin flip works
    if log_s_new > log_s_orig or p_coin(np.exp(log_s_new - log_s_orig)):
        decoder = proposed_decoder
        last_score = log_s_new

        if log_s_new > best_score:
            best_decoder = proposed_decoder.copy()
            best_score = log_s_new

return best_decoder

```

## 1.11 Running It

The text you are going to first encrypt and then decode is taken from [Winston Churchill's speech](#) to the House of Commons on 4 June 1940. Naturally, it contains upper case letters and punctuation that we have not included in our alphabet. In the cell below, the function `clean_string` strips out all the punctuation and replaces upper case letters by the corresponding lower case letters.

```
[25]: ogtext = """
I have, myself, full confidence that if all do their duty, if nothing is
    ↪neglected,
and if the best arrangements are made, as they are being made, we shall prove
    ↪ourselves
once again able to defend our Island home, to ride out the storm of war, and to
    ↪outlive
the menace of tyranny, if necessary for years, if necessary alone. At any rate,
    ↪that is
what we are going to try to do. That is the resolve of His Majesty's
    ↪Government-every man
of them. That is the will of Parliament and the nation.
"""
original_string = clean_string(ogtext)
encoding_cipher = random_decoder()
string_to_decode = decode_text(original_string, encoding_cipher)
string_to_decode
```

```
[25]: 'k fahb xqzbnj nojj stgnklbgbs yfay kn ajj lt yfbkr loyq kn gtyfkgv kz gbvjbsybl
agl kn yfb dbzy arragvxbgzyz arb xalb az yfbq arb dbkgv xalb ub zfajj crthb
torzbjhbz tgsb avakg adjb yt lbnbgf tor kzjagl ftxb yt rklb toy yfb zytrx tn
uar agl yt toyjkhb yfb xbgasb tn yqraggq kn gbsbzzarq ntr qbarz kn gbsbzzarq
ajtgb ay agq rayb yfay kz ufay ub arb vtkgv yt yrq yt lt yfay kz yfb rbztjhb tn
fkz xawbzyqz vthbrgxbgybhbrq xag tn yfbx yfay kz yfb ukjj tn carjkaxbgy agl yfb
gayktg'
```

### 1.11.1 8a) Finding the Decoder

Run the following cell to generate your decoder. You may have to run the cell a couple of times to get the correct decoder.

```
[26]: new_decoder = metropolis(string_to_decode, wp_bigram_mc, 10000)
```

```
Score: -2663    Guess: v yidj hosjka ackk lmuavxjulj qyiq va ik
Score: -1199    Guess: a hime vyderf furr cosfalesce thit af ir
Score: -1102    Guess: i have myself full confidence that if al
Score: -1077    Guess: i have myself full confidence that if al
Score: -1077    Guess: i have myself full confidence that if al
Score: -1077    Guess: i have myself full confidence that if al
Score: -1077    Guess: i have myself full confidence that if al
Score: -1077    Guess: i have myself full confidence that if al
```

```
Score: -1077      Guess: i have myself full confidence that if al
Score: -1077      Guess: i have myself full confidence that if al
```

Run the cell below to decode the entire string.

```
[27]: decode_text(string_to_decode, new_decoder)
```

```
[27]: 'i have myself full confidence that if all do their duty if nothing is neglected
and if the best arrangements are made as they are being made we shall prove
ourselves once again able to defend our island home to ride out the storm of
war and to outlive the menace of tyranny if necessary for years if necessary
alone at any rate that is what we are going to try to do that is the resolve of
his makestys governmentevery man of them that is the will of parliament and the
nation'
```

### 1.11.2 8b) Decoding Your Secret Message

`secret_text` contains your customized, secret message. Use MCMC to decode it. (Note: Even with correct implementation of previous parts, it may take a few runs to successfully decode the text due to randomness in the algorithm. Even then, it may fail to properly decode some letters, especially those which generally don't occur very frequently in English, such as *q* or *j*.)

```
[28]: secret_decoder = metropolis(secret_text, wp_bigram_mc, 10000)
decode_text(secret_text, secret_decoder)
```

```
Score: -2436      Guess: ybo seyroxwtpd npuot yoxy ex p dhlyexyp
Score: -1611      Guess: rye firneddltam sabet redr id a mucridra
Score: -1594      Guess: rye pirnettldaf samed retr it a fucritra
Score: -1594      Guess: rye pirnettldaf samed retr it a fucritra
Score: -1594      Guess: rye pirnettldaf samed retr it a fucritra
Score: -1578      Guess: rye firnettldam paced retr it a musritra
Score: -1578      Guess: rye firnettldam paced retr it a musritra
Score: -1578      Guess: rye firnettldam paced retr it a musritra
Score: -1418      Guess: the fitnessgram pacer test is a multista
Score: -1418      Guess: the fitnessgram pacer test is a multista
```

```
[28]: 'the fitnessgram pacer test is a multistage aerobic capacity test that
progressively gets more difficult as it continues the meter pacer test will
begin in seconds line up at the start the running speed starts slowly but gets
faster each minute after you hear this signal beep a single lap should be
completed each time you hear this sound ding remember to run in a straight line
and run as long as possible the second time you fail to complete a lap before
the sound your test is over the test will begin on the word start on your mark
get ready start source the fitnessgram pacer test'
```

## 1.12 Conclusion

What you have learned: - How to implement the Metropolis algorithm to decode text encrypted by a substitution code, with simplifying assumptions about spaces, capitalization, and punctuation -

MCMC works - Abstract concepts such as balance and convergence of  $n$ -step transition matrices are remarkable in their own right and also have powerful applications

### 1.13 Submission Instructions

Many assignments throughout the course will have a written portion and a code portion. Please follow the directions below to properly submit both portions.

#### 1.13.1 Written Portion

- Scan all the pages into a PDF. You can use any scanner or a phone using applications such as CamScanner. Please **DO NOT** simply take pictures using your phone.
- Please start a new page for each question. If you have already written multiple questions on the same page, you can crop the image in CamScanner or fold your page over (the old-fashioned way). This helps expedite grading.
- It is your responsibility to check that all the work on all the scanned pages is legible.
- If you used  $\text{\LaTeX}$  to do the written portions, you do not need to do any scanning; you can just download the whole notebook as a PDF via  $\text{\LaTeX}$ .

#### 1.13.2 Code Portion

- Save your notebook using **File > Save Notebook**.
- Generate a PDF file using **File > Save and Export Notebook As > PDF**. This might take a few seconds and will automatically download a PDF version of this notebook.
  - If you have issues, please post a follow-up on the general Lab 4 Ed thread.

#### 1.13.3 Submitting

- Combine the PDFs from the written and code portions into one PDF. [Here](#) is a useful tool for doing so.
- Submit the assignment to Lab 4 on Gradescope.
- **Make sure to assign each page of your pdf to the correct question.**
- **It is your responsibility to verify that all of your work shows up in your final PDF submission.**

If you are having difficulties scanning, uploading, or submitting your work, please read the [Ed Thread](#) on this topic and post a follow-up on the general Lab 4 Ed thread.

**1.14 We will not grade assignments that do not have pages selected for each question.**

$$5. \quad r(i, j) = \frac{s(j)}{s(i)}$$

$$r(i, j) < 1$$

$$\frac{s(j)}{s(i)} < 1$$

$$\log\left(\frac{s(j)}{s(i)}\right) < \log(1)$$

$$\log(s(j)) - \log(s(i)) < 0$$

$$\log(s(j)) < \log(s(i))$$

$\therefore$  a higher log score indicates that the decoded letter follows the natural English language pattern more closely. Since  $\log(s(j))$  is smaller than  $\log(s(i))$ , state  $i$  is more favorable (higher score), and is less likely to move to state  $j$ . Since the logarithmic function is monotonic, applying the logarithm preserves the inequality, allowing us to work with log scores without changing the comparison.