Exercises

Exercise 1. (*Spell Checker*) Write a program `spell_checker.py` that accepts words from standard input; looks up each word in the file `data/misspellings.txt` that maps misspelled words to their correct spellings; and if it exists (ie, is misspelled), writes the word to standard output along with the correct spelling.

```
>_ ~/workspace/project6
$ python3 spell_checker.py
Try nto to become a man of sucess but rather try to become a man of value. ~ Albert Einstein
<enter>
nto -> not
sucess -> success
<ctrl-d>
```

```
spell_checker.py
from instream import InStream
from symboltable import SymbolTable
import stdio


# Entry point.
def main():
    # Set inStream to an input stream built from the file 'data/misspellings.txt'.
    ...

    # Set lines to the list of lines read from inStream.
    ...

    # Set misspellings to a new symbol table object.
    ...

    for ... in ...:
        # For each line (of the form 'misspelling correction') in lines...

        # Set tokens to the list obtained by splitting line using the split() method from str.
        ...

        # Insert the pair tokens[0]/tokens[1] into misspellings.
        ...

    while ...:
        # As long as standard input is not empty...

        # Set word to a string read from standard input.
        ...

        # If word exists in misspellings, then it is misspelled. So write the word and the
        # correction to standard output, separated by the string '->'.
        ...


if __name__ == '__main__':
    main()
```

Exercise 2. (*Word Occurrences*) Write a program `word_occurrences.py` that accepts *filename* (str) as command-line argument and words from standard input; and writes to standard output the word along with the indices (ie, locations) where it appears in the file whose name is *filename* — writes "Word not found" if the word does not appear in the file.

```
>_ ~/workspace/project6
$ python3 word_occurrences.py data/Beatles.txt
dead
<enter>
dead -> [3297, 4118, 4145, 4197]
parrot
<enter>
Word not found
<ctrl-d>
```

word_occurrences.py

```python
from instream import InStream
from symboltable import SymbolTable
import stdio
import sys


# Entry point.
def main():
    # Accept filename (str) as command-line argument.
    ...

    # Set inStream to an input stream built from filename.
    ...

    # Set words to the list of strings read from inStream.
    ...

    # Set occurrences to a new symbol table object.
    ...

    for i, word in enumerate(...):
        # For each word (having index i) in words...

        # If word does not exist in occurrences, insert it with an empty list as the value.
        ...

        # Append i to the list corresponding to word in occurrences.
        ...

    while ...:
        # As long as standard input is not empty...

        # Set word to a string read from standard input.
        ...

        # If word exists in occurrences, write the word and the corresponding list to standard
        # output, separated by the string '->'. Otherwise, write the message 'Word not found'.
        if ...:
            ...
        else:
            ...


if __name__ == '__main__':
    main()
```

Problems

Goal The purpose of this project is to use a Markov chain to create a statistical model of a piece of English text and use the model to generate stylized pseudo-random text and decode noisy messages.

Perspective In the 1948 landmark paper A Mathematical Theory of Communication, Claude Shannon founded the field of information theory and revolutionized the telecommunications industry, laying the groundwork for today's Information Age. In this paper, Shannon proposed using a *Markov chain* to create a statistical model of the sequences of letters in a piece of English text. Markov chains are now widely used in speech recognition, handwriting recognition, information retrieval, data compression, and spam filtering. They also have many scientific computing applications including the genemark algorithm for gene prediction, the Metropolis algorithm for measuring thermodynamical properties, and Google's PageRank algorithm for Web search. In this assignment, we consider two variants: generating stylized pseudo-random text and decoding noisy messages.

Markov Model of Natural Language Shannon approximated the statistical structure of a piece of text using a simple mathematical model known as a *Markov model.* A Markov model of *order* 0 predicts that each letter in the alphabet occurs with a fixed probability. We can fit a Markov model of order 0 to a specific piece of text by counting the number of occurrences of each letter in that text, and using these frequencies as probabilities. For example, if the input text is `'gagggagaggcgagaaa'`, the Markov model of order 0 predicts that each letter is `'a'` with probability 7/17, `'c'` with probability 1/17, and `'g'` with probability 9/17 because these are the fraction of times each letter occurs. The following sequence of characters is a typical example generated from this model:

```
gaggcgagaagagaagaaagagagagaaagagaag...
```

A Markov model of order 0 assumes that each letter is chosen independently. This independence does not coincide with statistical properties of English text because there a high correlation among successive characters in a word or sentence. For example, `'w'` is more likely to be followed with `'e'` than with `'u'`, while `'q'` is more likely to be followed with `'u'` than with `'e'`.

We obtain a more refined model by allowing the probability of choosing each successive letter to depend on the preceding letter or letters. A Markov model of order k predicts that each letter occurs with a fixed probability, but that probability can depend on the previous k consecutive characters. Let a k-*gram* mean any string of k characters. Then for example, if the text has 100 occurrences of `'th'`, with 60 occurrences of `'the'`, 25 occurrences of `'thi'`, 10 occurrences of `'tha'`, and 5 occurrences of `'tho'`, the Markov model of order 2 predicts that the next letter following the 2-gram `'th'` is `'e'` with probability 3/5, `'i'` with probability 1/4, `'a'` with probability 1/10, and `'o'` with probability 1/20.

A Brute-Force Solution Claude Shannon proposed a brute-force scheme to generate text according to a Markov model of order 1:

> “To construct [a Markov model of order 1], for example, one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. It would be interesting if further approximations could be constructed, but the labor involved becomes enormous at the next stage.”

Your task in this project is to write a Python program to automate this laborious task in a more efficient way — Shannon’s brute-force approach is prohibitively slow when the size of the input text is large.

Problem 1. (*Markov Model Data Type*) Define a data type called `MarkovModel` in `markov_model.py` to represent a Markov model of order k from a given text string. The data type must support the following API:

MarkovModel	
`MarkovModel(text, k)`	constructs a Markov model `m` of order `k` from `text`
`m.order()`	returns the order of `m`
`m.kgram_freq(kgram)`	returns the number of occurrences of `kgram` in `m`
`m.char_freq(kgram, c)`	returns the number of times character `c` follows `kgram` in `m`
`m.rand(kgram)`	using `m`, finds and returns a random character following `kgram`
`m.gen(kgram, n)`	using `m`, builds and returns a string of length `n`, the first `k` characters of which is `kgram`

- *Constructor* To implement the data type, define two instance variables: an integer `_k` that stores the order of the Markov model, and a symbol table `_st` whose keys are all the k-grams from the given text. The value corresponding to each key (say *kgram*) in `_st` is a symbol table whose keys are the characters that follow *kgram* in the text, and the corresponding values are their frequencies. You may assume that the input text is a sequence of characters over the ASCII alphabet so that all values are between 0 and 127. The frequencies should be tallied as if the text were circular (i.e., as if it repeated the first k characters at the end). For example, if the text is `'gagggagaggcgagaaa'` and $k = 2$, then the symbol table `_st` should store the following information:

```
{
    'aa': {'a': 1, 'g': 1},
    'ag': {'a': 3, 'g': 2},
    'cg': {'a': 1},
    'ga': {'a': 1, 'g': 4},
    'gc': {'g': 1},
    'gg': {'a': 1, 'c': 1, 'g': 1}
}
```

  If you are careful enough, the entire symbol table can be built in just one pass through the circular text. Note that there is no reason to save the original text or the circular text as an attribute of the data type. That would be a grossly inefficient waste of space. Your `MarkovModel` object does not need either of these strings after the symbol table is built.

- *Order.* Return the order k of the Markov Model.

- *Frequency.* There are two frequency methods.
  - `kgram_freq(kgram)` returns the number of times *kgram* was found in the original text. Returns 0 when *kgram* is not found. Raises an error if *kgram* is not of length k.
  - `char_freq(kgram, c)` returns the number of times *kgram* was followed by the character c in the original text. Returns 0 when *kgram* or c is not found. Raises an error if *kgram* is not of length k.
- *Randomly generate a character.* Return a character. It must be a character that followed the *kgram* in the original text. The character should be chosen randomly, but the results of calling `rand(kgram)` several times should mirror the frequencies of characters that followed the *kgram* in the original text. Raise an error if *kgram* is not of length k or if *kgram* is unknown.
- *Generate pseudo-random text.* Return a string of length n that is a randomly generated stream of characters whose first k characters are the argument *kgram*. Starting with the argument *kgram*, repeatedly call `rand()` to generate the next character. Successive k-grams should be formed by using the most recent k characters in the newly generated text.

To avoid dead ends, treat the input text as a *circular string*: the last character is considered to precede the first character. For example, if $k = 2$ and the text is the 17-character string `'gagggagaggcgagaaa'`, then the salient features of the Markov model are captured in the table below:

		frequency of			probability that		
		next char			next char is		
kgram	freq	a	c	g	a	c	g
aa	2	1	0	1	1/2	0	1/2
ag	5	3	0	2	3/5	0	2/5
cg	1	1	0	0	1	0	0
ga	5	1	0	4	1/5	0	4/5
gc	1	0	0	1	0	0	1
gg	3	1	1	1	1/3	1/3	1/3
	17	7	1	9			

Note that the frequency of `'ag'` is 5 (and not 4) because we are treating the string as circular.

A *Markov chain* is a stochastic process where the state change depends on only the current state. For text generation, the current state is a k-gram. The next character is selected at random, using the probabilities from the Markov model. For example, if the current state is `'ga'` in the Markov model of order 2 discussed above, then the next character is `'a'` with probability 1/5 and `'g'` with probability 4/5. The next state in the Markov chain is obtained by appending the new character to the end of the k-gram and discarding the first character. A trajectory through the Markov chain is a sequence of such states. Shown below is a possible trajectory consisting of 9 transitions.

```
trajectory:          ga  -->  ag  -->  gg  -->  gc  -->  cg  -->  ga  -->  ag  -->  ga  -->  aa  -->  ag
probability for a:       1/5      3/5      1/3       0        1       1/5      3/5      1/5      1/2
probability for c:        0        0       1/3       0        0        0        0        0        0
probability for g:       4/5      2/5      1/3       1        0       4/5      2/5      4/5      1/2
```

Treating the input text as a circular string ensures that the Markov chain never gets stuck in a state with no next characters.

To generate random text from a Markov model of order k, set the initial state to k characters from the input text. Then, simulate a trajectory through the Markov chain by performing $n - k$ transitions, appending the random character selected at each step. For example, if $k = 2$ and $n = 11$, the following is a possible trajectory leading to the output `gaggcgagaag`:

```
trajectory:          ga  -->  ag  -->  gg  -->  gc  -->  cg  -->  ga  -->  ag  -->  ga  -->  aa  -->  ag
output:              ga       g        g        c        g        a        g        a        a        g
```

```
>_ ~/workspace/project6

$ python3 markov_model.py banana 2
an a
na b
na a
na -
<ctrl-d>
freq(an, a) = 2
freq(na, b) = 1
freq(na, a) = 0
freq(na) = 2
$ python3 markov_model.py gagggagaggcgagaaa 2
aa a
ga g
gg c
ag -
cg -
gc -
<ctrl-d>
freq(aa, a) = 1
freq(ga, g) = 4
freq(gg, c) = 1
freq(ag) = 5
freq(cg) = 1
freq(gc) = 1
```

Directions:

- Instance variables
  - Order of the Markov model, `_k` (`int`).
  - A symbol table to keep track of character frequencies, `_st` (`SymbolTable`) — keys are all the k-grams from the given text and the value corresponding to each key (say *kgram*) is a symbol table whose keys are the characters that follow *kgram* in the text, and the corresponding values are their frequencies.
- `MarkovModel(text, k)`
  - Initialize instance variables appropriately.
  - Construct circular text `circ_text` from `text` by appending the first `k` characters to the end; for example, if `text = 'gagggagaggcgagaaa'` and `k = 2`, then `circ_text = 'gagggagaggcgagaaaga'`.
  - For each `kgram` from `circ_text`, and the character `next_char` that immediately follows `kgram`, increment the frequency of `next_char` in `_st[kgram]` by 1; for the above example, `_st`, at the end of this step, should store the following information:

```
{
    'aa': {'a': 1, 'g': 1},
    'ag': {'a': 3, 'g': 2},
    'cg': {'a': 1},
    'ga': {'a': 1, 'g': 4},
    'gc': {'g': 1},
    'gg': {'a': 1, 'c': 1, 'g': 1}
}
```

- `m.order()`
  - Return the order of the Markov model.
- `m.kgram_freq(kgram)`
  - Return the frequency of `kgram`, which is simply the sum of the values of `_st[kgram]`.
- `m.char_freq(kgram, c)`
  - Return the number of times `c` immediately follows `kgram`, which is simply the value of `c` in `_st[kgram]`.
- `m.rand(kgram)`
  - Use `stdrandom.discrete()` to randomly select and return a character that immediately follows `kgram`.
- `m.gen(kgram, n)`
  - Initialize a variable `text` to `kgram`.
  - Perform `n - _k` iterations, where each iteration involves appending to `text` a random character obtained using `self.rand()` and updating `kgram` to the last `_k` characters of `text`.
  - Return `text`.

Problem 2. (*Random Text Generator*) Implement a program `text_generator.py` that accepts k (int) and n (int) as command-line arguments, reads the input text from standard input (for efficiency reasons, use `sys.stdin.read()` to read the text) and builds a Markov model of order k from the input text; then, starting with the k-gram consisting of the first k characters of the input text, writes to standard output n characters generated by simulating a trajectory through the corresponding Markov chain, followed by a new line. You may assume that the text has length at least k, and also that $n \geq k$.

```
>_ ~/workspace/project6

$ python3 text_generator.py 2 50 < data/input17.txt
gaggcgagaggcgagggaagaaaggaagagagagagaaggagagaggcga
```

Directions:

- Accept command-line arguments k (int) and n (int).
- Initialize *text* to text read from standard input using `sys.stdin.read()`.
- Create a Markov model using *text* and k.
- Use the model to generate a random text of length n and starting with the first k characters of *text*.
- Write the random text to standard output.

Problem 3. (*Noisy Message Decoder*) Imagine you receive a message where some of the characters have been corrupted by noise. We represent unknown characters by the `~` symbol (we assume we don't use `~` in our messages). Implement the method `model.replace_unknown()` in the `markov_model.py` data type that decodes a noisy message `corrupted` by replacing each `~` in it with the most likely character and returns the decoded message. You may assume that the unknown characters are at least k characters apart and also appear at least k characters away from the start and end of the message.

This maximum-likelihood approach doesn't always get it right, but it fixes most of the missing characters correctly. Here are some details on what it means to find the most likely replacement for each `~`. For each unknown character, you should consider all possible replacement characters. You want the replacement character that makes sense not only at the unknown position (given the previous characters) but also when the replacement is used in the context of the k subsequent known characters. You can compute the probability of each possbile replacement character (aka hypothesis) by multiplying the probabilities of generating each of the $k+1$ characters in sequence: the missing one, and the k subsequent ones.

Implement a program `fix_corrupted.py` that accepts k (int) s (str) as command-line arguments representing the model order and corrupt message, reads the input text from standard input (for efficiency reasons, use `sys.stdin.read()` to read the text), and writes to standard output the most likely original string.

```
>_ ~/workspace/project6

$ python3 fix_corrupted.py 3 "it w~s th~ bes~ of tim~s, i~ was ~he wo~st of~times." < data/bible.txt
it was the bese of times, it was the woest of times.
$ python3 fix_corrupted.py 4 "it w~s th~ bes~ of tim~s, i~ was ~he wo~st of~times." < data/bible.txt
it was the best of times, it was the worst of times.
```

Directions:

- Main idea behind `m.replace_unknown(corrupted)`

  When we fix a corrupted message, we have to look at each missing letter in the context of what comes before it and what comes after it. For example, let the corrupted message be `'it w~s th'`, $k = 4$, and let the characters that follow the 4-gram `'it w'` be `'a'`, `'b'`, and `'c'`. So we want to pick the best of three hypotheses (call them H_a, H_b, and H_c). Let's use the notation `'abcd'|'e'` to mean the probability of finding an `'e'` after the 4-gram `'abcd'`. This probability is 0 if `'e'` does not follow `'abcd'` in the text.

  The likelihood of H_a is the product of $(k+1)$ probabilities: `'it w'|'a'`, `'t wa'|'s'`, `' was'|' '`, `'was '|'t'`, and `'as t'|'h'`.

The likelihood of H_b is the product of the following $(k+1)$ probabilities: `'it w'|'b'`, `'t wb'|'s'`, `' wbs'|' '`, `'wbs '|'t'`, and `'bs t'|'h'`.

The likelihood of H_c is the product of the following $(k+1)$ probabilities: `'it w'|'c'`, `'t wc'|'s'`, `' wcs'|' '`, `'wcs '|'t'`, and `'cs t'|'h'`.

Now, the character that we use to replace `~` with is the one with the maximum likelihood. So if $\max(H_a, H_b, H_c) = H_a$, then we would replace `~` by the character `'a'`. We use the `_argmax()` helper function for this.

- Pseudocode for `m.replace_unknown(corrupted)`

```
if corrupted[i] == '~':
    kgram_before = kgram before ~
    kgram_after = kgram after ~
    probs = []
    for each hypothesis from hypotheses (ie, characters that can replace ~):
        context = kgram_before + hypothesis + kgram_after
        p = 1.0
        for i from 0 to _k + 1:
            kgram = kgram from context starting at i
            char = character from context that follows kgram
            if kgram or char is non-existent, then set p to 0 and break
            Otherwise, multiply p by probability of char following kgram
        append p to probs
    append to original the hypothesis that maximizes probs (use _argmax())
```

- Implement `fix_corrupted.py` as follows:
  - Accept *k* (int) and *corrupted* (str) as command-line arguments.
  - Initialize *text* to text read from standard input using `sys.stdin.read()`.
  - Create a Markov model using *text* and *k*.
  - Use the model to decode *corrupted*.
  - Write the decoded text to standard output.

Data Be sure to test your programs thoroughly using files provided under the `data` folder. For example

```
>_ ~/workspace/project6
$ python3 text_generator.py 5 50 < data/Beatles.txt
Words you. Baby you'll all you, yeh. It too long I
$ python3 fix_corrupted.py 3 "she s~lls sea s~ells on th~ sea s~ore" < data/wiki_100k.txt
she sells sea spells on the sea store
```

Acknowledgements This project is an adaptation of the Markov Model of Natural Language assignment developed at Princeton University by Robert Sedgewick and Kevin Wayne.

Files to Submit

1. `spell_checker.py`
2. `word_occurrences.py`
3. `markov_model.py`
4. `text_generator.py`
5. `fix_corrupted.py`
6. `report.txt`

Before you submit your files, make sure:

- You do not use concepts from sections beyond "Stacks, Queues, and Symbol Tables".
- Your programs meet the style requirements by running the following command in the terminal.

```
>_ ~/workspace/project6
$ pycodestyle <program>
```

- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You use the template file `report.txt` for your report.
- Your report meets the prescribed guidelines.