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*) Create a data type `MarkovModel` in `markov_model.py` to represent a Markov model of order k from a given text string. The data type must implement the following API:

method	description
`MarkovModel(text, k)`	create a Markov model `model` of order k from *text*
`model.order()`	order k of Markov model
`model.kgram_freq(kgram)`	number of occurrences of *kgram* in text
`model.char_freq(kgram, c)`	number of times that character c follows *kgram*
`model.rand(kgram)`	a random character following the given *kgram*
`model.gen(kgram, T)`	a string of length T characters generated by simulating a trajectory through the corresponding Markov chain, the first k characters of which is *kgram*

- *Constructor* To implement the data type, define two attributes: an integer `k` that stores the order of the Markov model, and a dictionary[1] `st` whose keys are all the k-grams from the given text. The value corresponding to each key (say *kgram*) in `st` is a dictionary 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 dictionary `st` should look like the following:

  ```
  {’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 dictionary 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 dictionary 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 T 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

[1] Dictionaries play a pivotal role in this project, so before you write any code, make sure you understand what dictionaries are and how they work. The dictionary methods such as `keys()`, `values()`, and `setdefault()` will come in handy.

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 $T - k$ transitions, appending the random character selected at each step. For example, if $k = 2$ and $T = 11$, 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
```

```
$ python markov_model.py
text: banana, k = 2
freq(’an’, ’a’)  = 2
freq(’na’, ’b’)  = 1
freq(’na’, ’a’)  = 0
freq(’na’)       = 2

text: one fish two fish red fish blue fish, k = 4
freq(’ish ’, ’r’) = 1
freq(’ish ’, ’0’) = 0
freq(’ish ’)      = 3
freq(’tuna’)      = 0

text: gagggagaggcgagaaa, k = 2
freq(’aa’, ’a’) = 1
freq(’ga’, ’g’) = 4
freq(’gg’, ’c’) = 1
freq(’ag’)      = 5
freq(’cg’)      = 1
freq(’gc’)      = 1
```

Problem 2. (*Random Text Generator*) Write a client program `text_generator.py` that takes two command-line integers k and T, 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, prints out T 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 $T \geq k$.

```
$ python text_generator.py 2 11 < data/input17.txt
gaaggcgagga
$ python text_generator.py 2 11 < data/input17.txt
gaagaagagag
```

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 given an order k Markov model and conditional on the surrounding text 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 perfect, 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. For example, we expect the unknown character in `"was ~he wo"` to be `'t'` and not simply the most likely character in the context of `"was "`. You can compute the probability of each hypothesis by multiplying the probabilities of generating each of $k + 1$ characters in sequence: the missing one, and the k next ones.

Write a client program `fix_corrupted.py` that takes an integer k (model order) and a string s (noisy message) as command-line arguments, reads the input text from standard input (for efficiency reasons, use `sys.stdin.read()` to read the text), and prints out the most likely original string.

```
Original    : it was the best of times, it was the worst of times.
Noisy       : it w~s th~ bes~ of tim~s, i~ was ~he wo~st of~times.
```

```
$ python fix_corrupted.py 4 "it w~s th~ bes~ of tim~s, i~ was ~he wo~st of~times." < data/obama.txt
it was the best of times, it was the worst of times.
$ python fix_corrupted.py 2 "it w~s th~ bes~ of tim~s, i~ was ~he wo~st of~times." < data/obama.txt
it was the best of times, is was the worst of times.
```

Data The `data` directory contains several input files. Make sure to test your client programs against them.

```
$ python text_generator.py 5 50 < data/obama.txt
Well, you the last those what we've got to Russian
```

```
$ python 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
```

Files to Submit

1. `markov_model.py`
2. `text_generator.py`
3. `fix_corrupted.py`
4. `report.txt`

Before you submit:

- Make sure your programs meet the input and output specifications by running the following command on the terminal:

  ```
  $ python run_tests.py [<problems>]
  ```

  where the optional argument `<problems>` lists the numbers of the problems you want to test; all the problems are tested if no argument is given.

- Make sure your programs meet the style requirements by running the following command on the terminal:

  ```
  $ pep8 <program>
  ```

  where `<program>` is the `.py` file whose style you want to check.

- Make sure your report doesn't exceed 400 lines, doesn't contain spelling mistakes, and doesn't contain lines that exceed 80 characters.

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