Problem Set 4

Handed out: March 6, 2017

Due: March 14, 2017 @ 11:59 PM

This problem set has two parts. They are not dependent on one another, so feel free to work on them in parallel, or out-of-order.

The first part, A, allows you to practice thinking about problems in a recursive fashion, taking advantage of the idea that one can reduce the problem to a simpler version of the same problem. In ps4a.py, you will write two recursive functions that take a data structure representing a tree as input and perform certain operations on the values stored in the tree.

The second part, B and C, will give you experience in thinking about problems in terms of classes, each instance of which contains specific attributes as well as methods for manipulating them. In ps4b.py, you will use object-oriented programming to write a Caesar/shift cipher. In ps4c.py, you will use object-oriented programming to write a very simple substitution cipher.

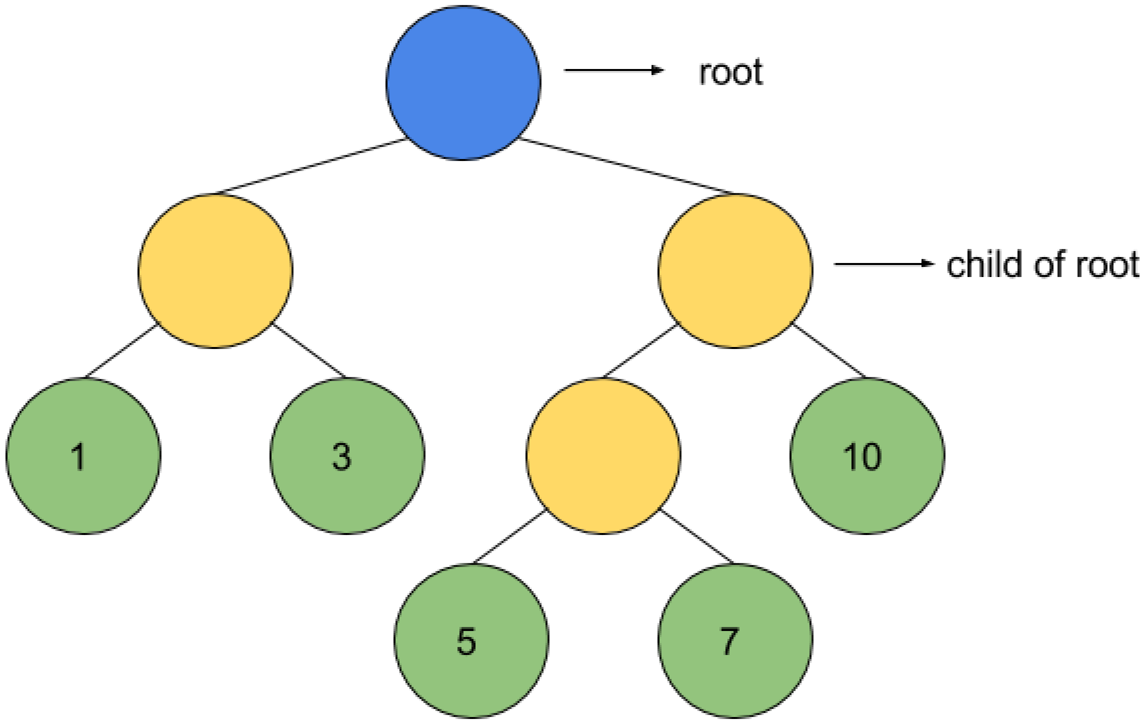
As always, please do not rename the files we provide you with, change any of the provided helper functions, change function/method names, or delete provided docstrings. You will need to keep **`words.txt`** and **`story.txt`** in the same folder in which you store **`ps4a.py`**, **`ps4b.py`** and **`ps4c.py`**.

Finally, please consult the Style Guide on Stellar, as we will be taking point deductions for violations (e.g. non-descriptive variable names and uncommented code). For this pset style guide numbers **6, 7 and 8** will be highly relevant so make sure you go over those before starting the pset, and again before you hand it in!

Part A

In this part, you will be working with recursive operations on trees.

A tree is a hierarchical data structure composed of linked nodes. The highest node is called the *root*, which has *branches* that link it to other nodes, which are themselves roots of their respective *subtrees*. A simple tree is shown below:



In this tree, the **root** is indicated by **blue**, the **intermediate nodes** (nodes that themselves have children) are indicated by **yellow**, and **leaves** (nodes without children) are indicated by **green**.

We can make a few observations for this specific tree:

- Leaves hold data: in this case, integers. We can also consider trees where branches also hold data, but we will restrict ourselves to leaves this time.
- Data is hierarchical: each node has a **parent** (except the root) and each node has **1 or more children** (except the leaves). We will be using this nomenclature in the rest of the problem set.
- Trees are inherently recursive: the right branch of the root (a **subtree**) is itself a tree.

Part A0: Data representation practice

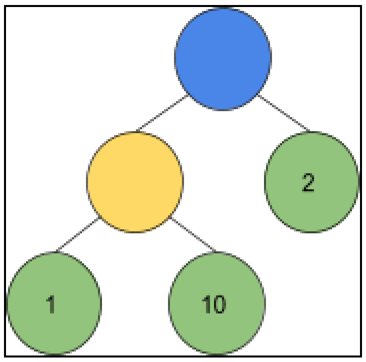
In this problem set, we will be using lists to represent trees. Since lists do not have hierarchical information themselves, we will be using sublists to indicate subtrees.

The tree above can be represented as the following list:

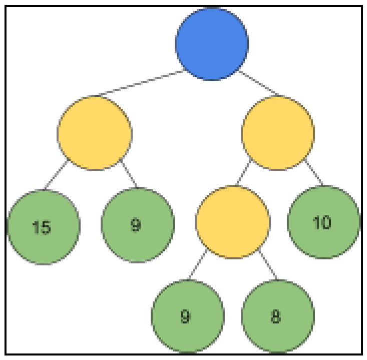
```
example_tree = [[1,3],[[5,7],10]]
```

We represent each leaf with the integer value it holds, and non-leaves (root or intermediate nodes) with a list. Note that the recursion property holds: if we consider the right branch, `[[5,7],10]`, we see that it is also a valid tree.

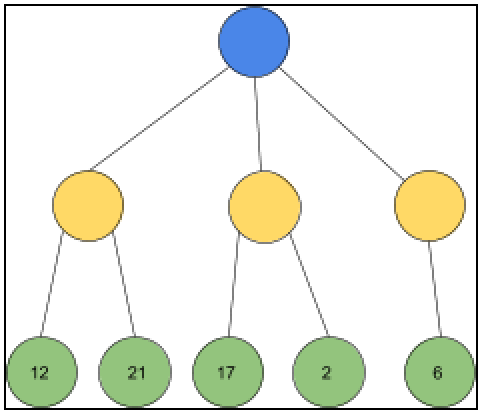
We will practice this notation in this part. For the trees shown below, create lists accurately representing the data. Put them into the variables at the top of ps4a.py, named **tree1**, **tree2** and **tree3**.



Tree 1



Tree 2



Tree 3

When you correctly initialize the three variables, the test named **`test_data_representation`** in test_ps4.py should pass. You can then proceed to the next part.

DO NOT PROCEED UNTIL YOU PASS THIS TEST. Tests for parts A1, A2 and A3 rely on the values of the trees you generate.

Part A1: Addition on tree leaves

Write a recursive function, `add_tree`, that adds up all the values of the leaves in each tree. **This function must be recursive; non-recursive implementations will receive a zero.**

Suggested approach:
In order to solve any recursive problem, we must have at least one base case and at least one recursive case. We can think of our base case as the simplest input we could have to the problem (for which determining the solution is trivial and requires no recursion).
For this approach, our base case is if a tree is empty, represented by an empty list `[]`.

If a tree is not empty, we need to identify a simpler version of the problem that, if solved, will help us find the sum for the entire tree. The pseudocode below gives one approach to recursively solving this problem.

Given an input tree, `T`:

Base case: If `T` is an empty list `[]`, the sum of its leaves would be 0.

Recursive cases: Both of these cases rely on the following observation: the sum of all leaves across all the root's children is equal to the sum of the leaves in the first child + the sum of the leaves across all the other children. To put it in algebra,

$$a + b + c + d = a + (b + c + d)$$

Case 1: The first child of `T` is a leaf. We could then get the sum of the entire tree by adding this leaf's value to the sum of the leaves of all of its siblings.

> Example:
> `T = [1,[2,5],[3,4]]]`
> The sum of T is the value of its first child (`1`) plus the sum of `[[2,5],[3,4]]` (which is just the original tree with the first child removed.)

Case 2: The first child of **`T`** is also a tree. We could then get the sum of the entire tree by adding the sum of the leaves of that child to the sum of the leaves of its siblings.

Example:
`T = [[1,2],[3,[4,5]]]`
The sum of T is the sum of the child tree **`[1,2] +`** the sum of the tree **`[[3,[4,5]]]`** (which again, is just the original tree with the first child removed.)

You should test your function using the variables from the previous part. For example:

```
add_tree(tree1) # should be 13
add_tree(tree2) # should be 51
add_tree(tree3) # should be 58
```

Print out the results of these function calls and make sure they are what you expect.

Now, your code should also pass the test **`test_add_example_trees`**.

Part A2: Arbitrary operations on tree leaves

Build upon your code from Part 1 to write a function, **`operate_tree`**, that can carry out arbitrary operations on the leaves of a tree. This function also has to be recursive, and it should take in three parameters: the tree **`tree`**, the operation to execute **`op`**, and the value that the function should return in base case (where the tree is empty), **`base_case.`**
Note that the second parameter, **`op`**, will be a function by itself- -- NOT the result of a function call.

Example operations are given below:

```
def addem(a, b):
    return a + b
```

```
def prod(a, b):
     return a * b
```

Example usage below. After you write your function, you should test it by printing out the results of these function calls:

```
operate_tree(tree1, addem, 0) # should be 13
operate_tree(tree1, prod, 1)  # should be 20
operate_tree(tree2, addem, 0) # should be 51
operate_tree(tree2, prod, 1)  # should be 97200
```

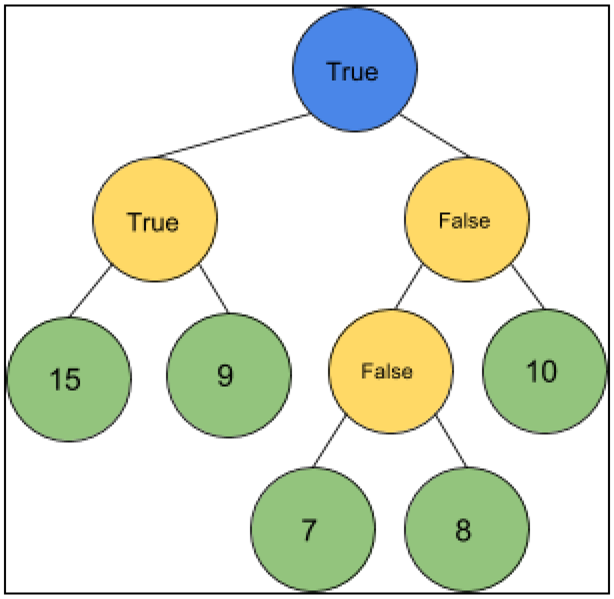
HINT! You should be able to reuse the structure of your **`add_tree`** function, only changing a few lines of code. Think about the places where you use addition in the previous function, and how you could change that function to use "**`addem`**" instead of "**`+`**". Then, think about how to generalize that to work with with any **`op`** and any **`base_case`** value.

You should now pass **`test_op_add_example_trees`** and **`test_op_prod_example_trees`**.

Part A3: Searching a tree

Now we will try writing our own operation function, similar to **`prod`** and **`addem`** from the last section. This time, the function is called **`search9`**. It should return **`True`** if the value "9" is in one of the tree's leaves, and **`False`** otherwise.

This operator will be slightly more complicated than **`addem`** and **`prod`** were, because it needs to be able to handle both booleans *and* integers as input. Consider the following tree:



At the non-leaf nodes, you can see the True/False decision, and therefore what the function **`search9`** should return, for each sub-tree. Looking at the left two yellow nodes, we see that **`search9(15, 9)`** should return True, while **`search9(7, 8)`** should return False.
However, we aren't always combining two numbers -- note that at the rightmost child of the root, we need to combine a boolean from the lower subtree with a number, 10. In that case, we'd be calling **`search9(False, 10).`**

Therefore, your function **`search9(a, b)`** should behave as follows:

- If **`a`** and **`b`** are both integers, return **`True`** if *either of them* are **`9`**.
- If **`a`** and **`b`** are both booleans, return **`True`** if *either of them* are **`True`**.
- If one of **`a`** and **`b`** is an integer, and the other is a boolean, return **`True`** if the boolean is **`True`** or the integer is **`9`** (or both).

You should now pass all of the tests in test_ps4a.py.

Part B: Cipher Like Caesar

Introduction

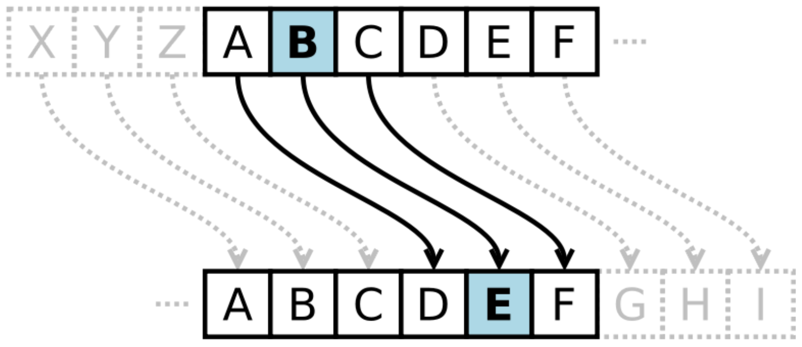
In order to prevent his important military correspondence from being intercepted and understood by the enemy, Julius Caesar used a simple trick that made the messages much harder for a human to read. In this problem set, we will write code that can both apply his technique to obscure a message, and translate an obscured message back into a readable form.

Here is some important vocabulary we will use going forward:

- **Encryption** - the process of obscuring or encoding messages to make them unreadable
- **Decryption** - converting encrypted messages back to their original, readable form
- **Cipher** - a technique or algorithm used for performing encryption and decryption
- **Plaintext** - the original, readable message
- **Ciphertext** - the encrypted message. *A ciphertext still contains all of the original message information, even though it looks like gibberish.*

How the Caesar Cipher Works

The idea of the Caesar Cipher (Wikipedia page here) is to pick an integer amount and shift every letter of your message by that amount.
Suppose the shift we choose is k. Then, all instances of the ith letter of the alphabet that appear in the plaintext should become the (i + k)th letter of the alphabet in the ciphertext. Consider the case where we use `k = 3`.This results in a mapping of letters as follows:



You will need to be careful to properly handle the case where the shift “wraps around” to the start of the alphabet, when `i+k > 26` (such as with X, Y, and Z in the diagram above.)

We will treat uppercase and lowercase letters separately, so that uppercase letters are always mapped to an uppercase letter, and lowercase letters are always mapped to a

lowercase letter. If an uppercase letter maps to “A”, then the same lowercase letter should map to “a”.
Punctuation, spaces, and numbers should stay the same as they were in the original message. For example, a plaintext message with a comma should have a corresponding ciphertext with a comma at the same position.

Plaintext	**Shift value, k**	**Ciphertext**
“abcdef”	2	“cdefgh”
“Hello, World!”	4	“Lipps, Asvph!”
“I am 19 years old”	3	“L dp 19 bgduv rog”
“......”	Any value	“......”
“” (empty string)	Any value	“”

Using Classes and Inheritance

This is your first experience coding with classes! Get excited! They are a powerful idea that you will use throughout the rest of your programming career.

For this problem set, we will use a parent class called **`Message`**, which has two child classes: **`PlaintextMessage`** and **`CipherTextMessage`**.

- **`Message`** contains methods that both plaintext and ciphertext messages will need to use -- for example, a method to get the text of the message. The child classes will inherit these shared methods from their parent.
- **`PlaintextMessage`** contains methods that are specific to a plaintext message, such as a method for encrypting a message given its shift value k.
- **`CiphertextMessage`** contains methods that are specific to a ciphertext, such as a method to decrypt a message.

When you have finished implementing these classes, you will be able to:

- Create a **`PlaintextMessage`** instance using a plaintext string, and then use a class method to get an encrypted version of the message,
- create a **`CiphertextMessage`** instance using an encrypted string, and then use a class method to try to decrypt it,
- Combine the two classes to encrypt and then decrypt a message, and then check that the new result matches your original message.

Your job will be to fill in methods for all three of these classes -- `Message`, `PlaintextMessage`, and `CiphertextMessage` -- according to the specifications given in the docstrings of ps4b.py.

There are a few helper functions we have implemented for you: **`load_words`**, **`is_word`**, and **`get_story_string`**. You may use these in your solution. You don't need to understand exactly how they work, but you should read the associated comments to understand what they do, and how to use them.

Part 1: Message

Fill in the methods of the Message class found in ps4b.py, according to the specifications in the docstrings.

We have provided skeleton code in the Message class for the following functions. Your task is to fill them in. Please see the docstring comment with each function for more information about the function's specification.

- `__init__(self, text)`
- `get_message_text(self)`
  This should return an immutable version of the message text we added to this object in init. Luckily, strings are already immutable objects, so we can simply return that string.
- `get_valid_words(self)`
  This should return a COPY of self.valid_words to prevent someone from accidentally mutating the original list.
- `build_shift_dict(self, shift)`
  You may find the string module's ascii_lowercase and ascii_uppercase constants helpful. Additionally, keep in mind the modulo operator, `%`, which returns the remainder when division is performed: for example,
  ```
  >>> 10 % 8
  2
  >>> 5 % 7
  5
  >>> 2 % 2
  0
  ```
- `apply_shift(self, shift_dict)`

Remember that spaces, punctuation and numbers should not be changed when the cipher is applied.

Side note: why do we write getters and setters?

Recall that you should never directly access attributes outside a class:

That's why we have *getter* and *setter* methods outlined in the skeleton code for you to implement, and to then use later on when you write tests. A getter method should just return an attribute of the object, and a setter method should just set an attribute of the object equal to the argument passed in. If you are writing more than 1-2 lines of code for any of these, you are are probably overthinking it!

Although they are simple, these methods are very important in object-oriented programming. We use them to ensure that we don't manipulate attributes we shouldn't be changing.
For example, if you you have created a Message `m`, you should never say `m.message_text`; instead, use `m.get_message_text()`.
Note that within the Message class (i.e. in its other class methods), it is fine to say `self.message_text`.
Directly using class attributes **outside** of a class itself (instead of using the getters and setters) will result in a point deduction on the problem set – and more importantly, may cause you headaches as you design and implement class hierarchies.

You can test out your code so far by running test_ps4b.py. Make sure it's in the same folder as your problem set. At this point, your code should be able to pass all the tests beginning with "`test_message`", but not the ones beginning with "`test_plaintext_message`" or "`test_ciphertext_message`" -- we will implement code for those in the next section.

Part 2: PlaintextMessage

Fill in the methods of the PlaintextMessage class found in ps4b.py according to the specifications in the docstrings.

- `__init__(self, text, shift)`
  You should use the parent class constructor in this method to make your code more concise. Take a look at Style Guide #7 if you are confused.
- `get_shift(self)`
- `get_encryption_dict(self)`
  Note: this should return a COPY of self.encryption_dict to prevent someone from mutating the original dictionary.
- `get_encrypted_message_text(self)`
- `change_shift(self, shift)`
  Think about what other methods you can use to make this easier. It shouldn't take more than a couple lines of code.

You can test your new class by running test_ps4b.py. You should now be able to pass all the tests in that file, except for `test_ciphertext_decrypt_message` and `test_valid_word_checking`.

Part 3: CiphertextMessage

Given an encrypted message, if you know the shift used to encode the message, decoding it is trivial. If `message` is the encrypted message, and `s` is the shift used to encrypt the message, then applying a shift of `26-s` gives you the original plaintext message.

The problem is that you won't be given the shift along with the encrypted message. But fortunately, our encryption method only has 26 distinct possible values for the shift. Since we know the messages are written in English, if we write a program that tries each shift and maximizes the number of valid English words in the decoded message, we can decrypt the ciphertext.

Fill in the methods of the CiphertextMessage class found in ps4b.py according to the specifications in the docstrings.

- `__init__(self, text)`
  As with PlaintextMessage, use the parent class constructor to make your code more concise. Take a look at Style Guide #7 if you are confused.
- `decrypt_message(self)`

You may find the helper function `is_word(wordlist, word)` and the string method `split` useful. Note: `is_word` will ignore punctuation and other special characters when considering whether a word is valid.

You should now be able to pass **all** the tests in test_ps4b.py.

Part 4: Writing Your Own Tests

Write two test cases for `PlaintextMessage` and two test cases for `CiphertextMessage` in the functions `test_plaintext_message()` and `test_ciphertext_message()`, at the bottom of ps4b.py. The function comments contain some sample tests to get you started.

Each test case should display the **input**, **expected output**, and **actual output**. Each case should handle a different case for the inputs: for example, you might write one to test that messages with capital letters are correctly encoded. Put a comment above each test case you write explaining what it is testing. *Remember the importance of using getters and setter functions!*

Run your test cases by uncommenting the specified lines at the bottom of the file under `if __name__ == '__main__'`.

Now, try out using your classes to decode the file story.txt. Write the code to do this inside the function `decode_story()`.

Hint: The skeleton code contains a helper function `get_story_string` that returns the encrypted version of the story as a string. Create a `CiphertextMessage` object using the story string, and then use `decrypt_message` to return the appropriate shift value and unencrypted story.

Part C: Substitution Cipher

A more secure way to encrypt your messages is to use a substitution cipher. In this approach, rather than shifting letters by a fixed amount, you substitute a randomly selected letter for each original letter. For the letter “a”, you could pick any of the 26 letters (including keeping “a”), for the letter “b”, you could then pick any of the remaining

25 letters (all but the one selected for “a”) and so on.

The number of possibilities to test if you wanted to decrypt a substitution ciphered message is much larger than for a Caesar cipher (26! compared to 26), making it very difficult to try all possible shift values as we did to decrypt messages in part B. So for this problem, we are going to just consider substitution ciphers in which the vowels are encoded and the consonants remain the same. As with the Caesar Cipher, lowercase and uppercase versions of a letter are mapped to the same letters: for example, if ‘A’ maps to ‘O’ then ‘a’ maps to ’o’.

Using Classes and Inheritance

Similar to the Caesar cipher, we are going to use classes to explore this idea. We will have a SubMessage class with general functions for handling Substitution Messages of this kind. We will also write a class with a more specific implementation and specification, EncryptedSubMessage, that inherits from the SubMessage class.

Your job will be to fill methods for both classes according to the specifications given in the docstrings of ps4c.py. As in part B, remember the importance of using getter and setter methods rather than directly accessing attributes.

We have provided two of the same helper functions as in part B - `load_words` and `is_word`. We have also provided a new helper function, `get_permutations` -- more on that later.

Part 1: SubMessage

Fill in the methods of the SubMessage class found in ps4c.py according to the specifications in the docstrings.

We have provided skeleton code for the following methods in the SubMessage class - your task is to implement them. Please see the docstring comment with each function for more information about the function specification.

- `__init__(self, text)`
- `get_message_text(self)`
- `get_valid_words(self)`

Note: this should return a COPY of self.valid_words to prevent someone from mutating the original list.

- `build_transpose_dict(self, vowels_permutation)`
- `apply_transpose(self, transpose_dict)`
  Pay close attention to the input specification when testing this function.

You can test your code by running test_ps4c.py. You should now pass all of the tests in that file **except** `test_encrypted_submessage_decrypt_message`.

Part 2: EncryptedSubMessage

Fill in the methods of the EncryptedSubMessage class found in ps4c.py according to the specifications in the docstrings.

Given an encrypted message, if you know the substitutions used to encode the message, decoding it is trivial. You could just replace each letter with the correct, decoded letter. The problem, of course, is that you don't know the substitution. Even though our substitution cipher keeps all the consonants the same, we still have a lot of options for trying different substitutions for the vowels.

Fortunately, as in Part B, we can try applying a possible substitution and seeing if the resulting words are real English words. We then select the decryption that yields the most valid English words.

Note that we have provided constants containing the values of the uppercase vowels, lowercase vowels, uppercase consonants, and lowercase consonants, for your convenience.

We have also provided a helper function, `get_permutations`, that provides all possible unique permutations (that is, possible orderings) of the characters in an input string. For example, the possible permutations of 'abc' are 'abc', 'acb', 'bac', 'bca', 'cab', 'cba'. Note that a string is a (trivial) permutation of itself. See the function's docstring for more details; you do not need to understand how it works internally, but you should understand what it does and how to use it.

The methods you should fill in are:

- `__init__(self, text)`
  Hint: use the parent class constructor to make your code more concise. Take a look at Style Guide #7 if you are confused.
- `decrypt_message(self)`

You should now pass all of the tests in test_ps4c.py.

Part 3: Writing Your Own Tests

Write two test cases for `SubMessage` and two test cases for `EncryptedSubMessage` in the functions `test_submessage()` and `test_encrypted_submessage()`, at the bottom of ps4b.py. The function comments contain some sample tests to get you started.

Each test case should display the **input**, **expected output**, and **actual output**. Each case should handle a different case for the inputs: for example, you might write one to test that messages with capital letters are correctly encoded. Put a comment above each test case you write explaining what it is testing.

Run your test cases by uncommenting the specified lines at the bottom of the file under `if __name__ == '__main__'`.

Hand-In Procedure

1. Save

Save your solutions with the original file names: **ps4a.py, ps4b.py, ps4c.py**. *Do not ignore this step, or save your file(s) with a different name!* Make sure your code runs without throwing syntax errors, and passes all of the provided test cases in test_ps4b.py, and test_ps4c.py. Make sure to delete any debugging print statements or commented-out sections of code.

2. Time and Collaboration Info

At the start of each file, fill out the comment section with the number of hours (roughly) you spent on the problems in that part, the names of the people you collaborated with, and number of late days you used, if any. For example:

```
# Problem Set 4C
# Name: Ana Bell
# Collaborators: John Guttag, Eric Grimson
# Time Spent: 3:30
# Late Days Used: 1
```

3. Submit

To submit a file, upload it to the section for Problem Set 4 on the submission portal. You may upload new versions of each file until the **11:59pm** deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.