Problem Set #4

Handed out: Lecture 7
Pseudocode: 11:59pm, Lecture 8. No late days can be used for this part of the assignment.
Due: 11:59pm, Lecture 10.

Pseudocode Solutions

Check your pseudocode against ours before you finish your implementation!

Introduction

Encryption is the process of obscuring information to make it unreadable without special knowledge. For centuries, people have devised schemes to encrypt messages — some better than others — but the advent of the computer and the Internet revolutionized the field. These days, it's hard not to encounter some sort of encryption, whether you are buying something online or logging into Athena.

A cipher is an algorithm for performing encryption (and the reverse, decryption). The original information is called plaintext. After it is encrypted, it is called ciphertext. The ciphertext message contains all the information of the plaintext message, but it's not in a format readable by a human or computer without the proper mechanism to decrypt it; it should resemble random gibberish to those not intended to read it.

A cipher usually depends on a piece of auxiliary information, called a key. The key is incorporated into the encryption process; the same plaintext encrypted with two different keys should have two different ciphertexts. Without the key, it should be difficult to decrypt the resulting ciphertext into readable plaintext.

This assignment will deal with a well-known (though not very secure) encryption method called the Caesar cipher. In this problem set you will need to devise your own algorithms and will practice using recursion to solve a non-trivial problem.

Caesar Cipher

In this problem set, we will examine the Caesar cipher. The basic idea in this cipher is that you pick an integer for a key, and shift every letter of your message by the key. For example, if your message was "hello" and your key was 2, "h" becomes "j", "e" becomes "g", and so on. If you're interested in learning more about the Caesar cipher, check out the Wikipedia article.

In this problem set, we will use a variant of the standard Caesar cipher where the space character is included in the shifts: space is treated as the letter after "z", so with a key of 2, "y" would become " ", "z" would become "a", and " " would become "b".

Getting Started

- ps4-psuedo.txt: For problems 2a and 4a.
- ps4.py: the skeleton you'll fill in
- words.txt: a list of English words
- fable.txt: an encoded fable

Run the code without making any modifications to it, in order to ensure that everything is set up correctly. The code that we have given you loads a list of words from a file. If everything is okay, after a small delay, you should see the following printed out:

```
Loading word list from file...
55902 words loaded.
```

If you see an IOError instead (e.g., No such file or directory), you should change the value of the WORDLIST_FILENAME constant (defined near the top of the file) to the complete pathname for the file words.txt (this will vary based on where you saved the file).

The file, ps4.py, has a few functions already implemented that you can use while writing up your solution. You can ignore the code between the following comments, though you should read and understand everything else:

```
# -----------------------------------
# Helper code
# (you don't need to understand this helper code)
. . .
# (end of helper code)
# -----------------------------------
```

Pseudocode

Pseudocode is writing out the algorithm/solution in a form that is like code, but not quite code. Pseudocode is language independent, uses plain English (or your native language), and is readily understandable. Algorithm related articles in wikipedia often use pseudocode to explain the algorithm.

Think of writing pseudocode like you would explain it to another person – it doesn't generally have to conform to any particular syntax as long as what's happening is clear to the grader.

Pseudocode is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and subroutines. The purpose of using pseudocode is that it is easier for humans to understand than conventional programming language code, and that it is a compact and

environment-independent description of the key principles of an algorithm. No standard for pseudocode syntax exists, as a program in pseudocode is not an executable program. – wikipedia

In order to help you solve these problems correctly, we are requiring that you submit **pseudocode** for your solutions to Problems 2 and 4 by **Tuesday**. To do this, read problems 2 and 4, and think about **high level algorithms** to solve both problems. Write down the steps in your algorithms and save it in a plain text file named ps4.txt. Upload this file to your workspace.

On Wednesday, we will post our own pseudocode. You can use our pseudocode or your own (if it's close enough), to write the Python code that actually solves problems 2 and 4.

Problem 1. Encryption and Decryption

Write a program to encrypt plaintext into ciphertext using the Caesar cipher. We have provided skeleton code for the following functions:

```
def build_coder(shift):
    """
    Returns a dict that can apply a Caesar cipher to a letter.
    The cipher is defined by the shift value. Ignores non-letter characters
    like punctuation and numbers.

    shift: -27 < int < 27
    returns: dict

    Example:     >>> build_coder(3)
    {' ': 'c', 'A': 'D', 'C': 'F', 'B': 'E', 'E': 'H', 'D': 'G', 'G': 'J',
    'F': 'I', 'I': 'L', 'H': 'K', 'K': 'N', 'J': 'M', 'M': 'P', 'L': 'O',
    'O': 'R', 'N': 'Q', 'Q': 'T', 'P': 'S', 'S': 'V', 'R': 'U', 'U': 'X',
    'T': 'W', 'W': 'Z', 'V': 'Y', 'Y': 'A', 'X': ' ', 'Z': 'B', 'a': 'd',
    'c': 'f', 'b': 'e', 'e': 'h', 'd': 'g', 'g': 'j', 'f': 'i', 'i': 'l',
    'h': 'k', 'k': 'n', 'j': 'm', 'm': 'p', 'l': 'o', 'o': 'r', 'n': 'q',
    'q': 't', 'p': 's', 's': 'v', 'r': 'u', 'u': 'x', 't': 'w', 'w': 'z',
    'v': 'y', 'y': 'a', 'x': ' ', 'z': 'b'}
    (The order of the key-value pairs may be different.)
    """
    ### TODO.
def build_encoder(shift):
    """
    Returns a dict that can be used to encode a plain text. For example, you
    could encrypt the plain text by calling the following commands
    >>>encoder = build_encoder(shift)
    >>>encrypted_text = apply_coder(plain_text, encoder)
    The cipher is defined by the shift value. Ignores non-letter characters
    like punctuation and numbers.

    shift: 0 <= int < 27
    returns: dict

    Example:     >>> build_encoder(3)
    {' ': 'c', 'A': 'D', 'C': 'F', 'B': 'E', 'E': 'H', 'D': 'G', 'G': 'J',
    'F': 'I', 'I': 'L', 'H': 'K', 'K': 'N', 'J': 'M', 'M': 'P', 'L': 'O',
    'O': 'R', 'N': 'Q', 'Q': 'T', 'P': 'S', 'S': 'V', 'R': 'U', 'U': 'X',
```

```
    'T': 'W', 'W': 'Z', 'V': 'Y', 'Y': 'A', 'X': ' ', 'Z': 'B', 'a': 'd',
    'c': 'f', 'b': 'e', 'e': 'h', 'd': 'g', 'g': 'j', 'f': 'i', 'i': 'l',
    'h': 'k', 'k': 'n', 'j': 'm', 'm': 'p', 'l': 'o', 'o': 'r', 'n': 'q',
    'q': 't', 'p': 's', 's': 'v', 'r': 'u', 'u': 'x', 't': 'w', 'w': 'z',
    'v': 'y', 'y': 'a', 'x': ' ', 'z': 'b'}
    (The order of the key-value pairs may be different.)
    HINT : Use build_coder.
    """
    ### TODO.
def build_decoder(shift):
    """
    Returns a dict that can be used to decode an encrypted text. For example,
you
    could decrypt an encrypted text by calling the following commands
    >>>encoder = build_encoder(shift)
    >>>encrypted_text = apply_coder(plain_text, encoder)
    >>>decrypted_text = apply_coder(plain_text, decoder)
    The cipher is defined by the shift value. Ignores non-letter characters
    like punctuation and numbers.

    shift: 0 <= int < 27
    returns: dict

    Example:     >>> build_decoder(3)
    {' ': 'x', 'A': 'Y', 'C': ' ', 'B': 'Z', 'E': 'B', 'D': 'A', 'G': 'D',
    'F': 'C', 'I': 'F', 'H': 'E', 'K': 'H', 'J': 'G', 'M': 'J', 'L': 'I',
    'O': 'L', 'N': 'K', 'Q': 'N', 'P': 'M', 'S': 'P', 'R': 'O', 'U': 'R',
    'T': 'Q', 'W': 'T', 'V': 'S', 'Y': 'V', 'X': 'U', 'Z': 'W', 'a': 'y',
    'c': ' ', 'b': 'z', 'e': 'b', 'd': 'a', 'g': 'd', 'f': 'c', 'i': 'f',
    'h': 'e', 'k': 'h', 'j': 'g', 'm': 'j', 'l': 'i', 'o': 'l', 'n': 'k',
    'q': 'n', 'p': 'm', 's': 'p', 'r': 'o', 'u': 'r', 't': 'q', 'w': 't',
    'v': 's', 'y': 'v', 'x': 'u', 'z': 'w'}
    (The order of the key-value pairs may be different.)
    HINT : Use build_coder.
    """
    ### TODO.
def apply_coder(text, coder):
    """
    Applies the coder to the text. Returns the encoded text.

    text: string
    coder: dict with mappings of characters to shifted characters
    returns: text after mapping coder chars to original text

    Example:
    >>> apply_coder("Hello, world!", build_encoder(3))
    'Khoor,czruog!'
    >>> apply_coder("Khoor,czruog!", build_decoder(3))
    'Hello, world!'
    """
    ### TODO.
def apply_shift(text, shift):
    """
    Given a text, returns a new text Caesar shifted by the given shift
    offset. The empty space counts as the 27th letter of the alphabet,
    so spaces should be replaced by a lowercase letter as appropriate.
    Otherwise, lower case letters should remain lower case, upper case
```

```
    letters should remain upper case, and all other punctuation should
    stay as it is.

    text: string to apply the shift to
    shift: amount to shift the text
    returns: text after being shifted by specified amount.

    Example:
    >>> apply_shift('This is a test.', 8)
    'Apq hq hiham a.'
    """
    ### TODO.
```

Once you've written this function, you should be able to use it to encode strings.

Problem 2. Code-breaking

Your friend, who is also taking 6.00, is really excited about the program she wrote for Problem 1 of this problem set. She sends you emails, but they're all encrypted with the Caesar cipher!

The problem is, you don't know which shift key she is using. The good news is, you know your friend only speaks and writes English words. So if you can write a program to find the decoding that produces the maximum number of words, you can probably find the right decoding (There's always a chance that the shift may not be unique. Accounting for this would probably use statistical methods that we won't require of you.)

Part a: Pseudocode

Think about an algorithm you could use to solve this problem. Write the steps down and save in the textfile named ps4.txt.

Part b: Python code

Implement find_best_shift. This function takes a wordlist and a bit of encrypted text and attempts to find the shift that encoded the text. A simple indication of whether or not the correct shift has been found is if all the words obtained after a shift are valid words. Note that this only means that all the words obtained are actual words. It is possible to have a message that can be decoded by two separate shifts into different sets words. While there are various strategies for deciding between ambiguous decryptions, for this problem we are only looking for a simple solution.

To assist you in solving this problem, we have provided a helper function: *is_word(wordlist, word)*. This simply determines if word is a valid word according to wordlist. This function ignores capitalization and punctuation.

Hint: You may find the function string.split to be useful for dividing the text up into words.

```
def find_best_shift(wordlist, text):
```

```
    """
    Decrypts the encoded text and returns the plaintext.

    text: string
    returns: 0 <= int 27

    Example:     >>> s = apply_coder('Hello, world!', build_encoder(8))
    >>> s
    'Pmttw,hdwztl!'
    >>> find_best_shift(wordlist, s) returns
    8
    >>> apply_coder(s, build_decoder(8)) returns
    'Hello, world!'
    """
    ### TODO
```

Once you've written this function you can decode your friend's emails!

Problem 3. Multi-level Encryption & Decryption

Clearly the basic Caesar cipher is not terribly secure. To make things a little harder to crack, you will now implement a multi-level Caesar cipher. Instead of shifting the entire string by a single value, you will perform additional shifts at specified locations throughout the string. This function takes a string text and a list of tuples shifts. The tuples in shifts represent the location of the shift, and the shift itself. For example a tuple of (0,2) means that the shift starts are position 0 in the string and is a Caesar shift of 2. Additionally, the shifts are layered. This means that a set of shifts [(0,2), (5, 3)] will first apply a Caesar shift of 2 to the entire string, and then apply a Caesar shift of 3 starting at the 6th letter in the string.

To do this, implement the following function according to the specification.

```
def apply_shifts(text, shifts):
    """
    Applies a sequence of shifts to an input text.

    text: A string to apply the Ceasar shifts to
    shifts: A list of tuples containing the location each shift should
    begin and the shift offset. Each tuple is of the form (location,
    shift) The shifts are layered: each one is applied from its
    starting position all the way through the end of the string.
    returns: text after applying the shifts to the appropriate
    positions

    Example:
    >>> apply_shifts("Do Androids Dream of Electric Sheep?", [(0,6), (3, 18),
(12, 16)])
    'JufYkaolfapxQdrnzmasmRyrpfdvpmEurrb?'
    """
    ### TODO.
```

Problem 4. Multi-level Code-breaking

Your friend has sent you another message, but this one can't be decrypted by your solution to Problem 2 — it must be using a multi-layer shift.

To keep things from getting too complicated, we will add the restriction that a shift can begin only at the start of a word. This means that once you have found the correct shift at one location, it is guaranteed to remain correct at least until the next occurrence of a space character.

Part a: Pseudocode

As in Problem 2, Part b, we want you to sketch out a high level step-by-step algorithm for solving this problem. HINT: Use recursion. Save your steps in ps4.txt and upload this to your workspace.

Part b: Python code

To do this, implement the following function according to the specification.

```
def find_best_shifts(wordlist, text):
    """
    Given a scrambled string, returns a shift key that will decode the text
to
    words in wordlist, or None if there is no such key.

    ***HINT: Make use of the recursive function
    find_best_shifts_rec(wordlist, text, start)***

    wordlist: list of words
    text: scambled text to try to find the words for
    returns: list of tuples.  each tuple is (position in text, amount of
shift)

    Examples:
    >>> s = random_scrambled(wordlist, 3)
    >>> s
    'eqorqukvqtbmultiform wyy ion'
    >>> shifts = find_best_shifts(wordlist, s)
    >>> shifts
    [(0, 25), (11, 2), (21, 5)]
    >>> apply_shifts(s, shifts)
    'compositor multiform accents'
    >>> s = apply_shifts("Do Androids Dream of Electric Sheep?", [(0,6), (3,
18), (12, 16)])
    >>> s
    'JufYkaolfapxQdrnzmasmRyrpfdvpmEurrb?'
    >>> shifts = find_best_shifts(wordlist, s)
    >>> print apply_shifts(s, shifts)
    Do Androids Dream of Electric Sheep?
    """
```

To solve this problem successfully, we highly recommend that you use recursion (did we say use recursion again?). The non-recursive version of this function is much more difficult to understand and code. The key to getting the recursion correct is in understanding the seemingly

unnecessary parameter “start”. As always with recursion, you should begin by thinking about your base case, the simplest possible sub-problem you will need to solve. What value of start would make a good base case? (Hint: the answer is NOT zero.)

To help you test your code, we’ve given you two simple helper functions:
random_string(wordlist, n) generates n random words from wordlist and returns them in a string.

random_scrambled(wordlist, n) generates n random words from wordlist and returns them in a string after encrypting them with a random multi-level Caesar shift. You can start by making sure your code decrypts a single word correctly, then move up to 2 and higher.

NOTE: This function depends on your implementation of apply_shifts, so it will not work correctly until you have completed Problem 3.

```
def find_best_shifts_rec(wordlist, text, start):
    """
    Given a scrambled string and a starting position from which
    to decode, returns a shift key that will decode the text to
    words in wordlist, or None if there is no such key.
    Hint: You will find this function much easier to implement
    if you use recursion.

    wordlist: list of words
    text: scambled text to try to find the words for
    start: where to start looking at shifts
    returns: list of tuples.  each tuple is (position in text, amount of
shift)
    """
    ### TODO.
```

Problem 5. The Moral of the Story

Now that you have all the pieces to the puzzle, please use them to decode the file, fable.txt. At the bottom of the skeleton file, you will see a method ***get_fable_string()*** that will return the encrypted version of the fable. Create the following method and include as a comment at the end of the problem set how the fable relates to your education at MIT.

```
def decrypt_fable():
     """
    Using the methods you created in this problem set,
    decrypt the fable given by the function get_fable_string().
    Once you decrypt the message, be sure to include as a comment
    at the end of this problem set how the fable relates to your
    education at MIT.

    returns: string - fable in plain text
    """
    ### TODO.
```

Hand-In Procedure

1. Save

You should be using ps4.txt to save your pseudocode answers. Remember, this part is turned in after Lecture 8!

You should be using the ps4.py skeleton given to you in this problem set. Fill in the code for the functions: *build_coder(), apply_coder(), apply_shift(), find_best_shift(), apply_shifts(),* and *find_best_shifts()*. Any other code is not necessary. Save your solution as ps4.py. Do not ignore this step or save your file with a different name.

2. Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people you collaborated with. For example:

```
# Problem Set 4
# Name: Jane Lee
# Collaborators (Discussion): John Doe
# Collaborators (Identical Solution): Jane Smith
# Time: 1:30
#
... your code goes here ...
```

6.00SC Introduction to Computer Science and Programming
Spring 2011