Homework 5

Due 11:59pm, Wednesday, March 6, 2019

Installation and Handin

Homework Setup. For each homework assignment, there may be support files that you will need to complete the assignment. These can be copied to your home directory by using the `cs4_install` command in a Brown CS Terminal window. For this homework type:

```
cs4_install hw05
```

There should now be a `hw05` folder within your homeworks directory. Using Terminal, you can move into the `hw05` folder with the cd command:

```
cd ~/course/cs0040/homeworks/hw05
```

Homework hand-in. Be sure to turn in all the files requested and that they are named exactly as specified, including spelling and case. When you're ready to submit the files, run:

```
cs4_handin hw05
```

from a Brown CS Terminal window from your `~/course/cs0040/homeworks/hw05` directory. The entire contents of your `~/course/cs0040/homeworks/hw05` directory will be handed in. Check for a confirmation email to ensure that your assignment was correctly submitted using the `cs4_handin` command. You can resubmit this assignment using the `cs4_handin` command at any time, but be careful, as only your most recent submission with be graded.

Note: Harder parts of this problem set are indicated by *. Problems that may be challenging are also indicate by * following their name.

Part I: Efficiency* (20)

Place your answers for Part I in a file named `hw05_1.py`.

Problem 5.1a) Power Function

In this problem, you will compare the efficiency of two different recursive versions of the power function. In lecture, we covered `power(b,n)`, which uses a recursive step based on the following definition:

$$b^n = \begin{cases} 1 & \text{if } n < 1 \\ b(b^{n-1}) & \text{otherwise} \end{cases}$$

In `hw05_1.py`, write the function `myPower(b,n)` based on the following definition:

$$b^n = \begin{cases} 1 & \text{if } n < 1 \\ (b^{n/2})^2 & n \text{ even} \\ b(b^{n-1}) & n \text{ odd} \end{cases}$$

You may assume `n` is a non-negative integer. As always, use test first design as you create a recursive version of myPower (and be sure to include all your test cases). For some guidance, this is how you could implement the even case:

```
sqrt = myPower(b, n/2)
return sqrt*sqrt
```

And this is how you could implement the odd case:

```
return b*myPower(b, n-1)
```

Problem 5.1b) Thoughtful Comparison*

Include your answers to the following question inside of a triple quoted string in `hw05_1.py`.

1) How many recursive calls are made when `power(2,9)` is evaluated? How about in the method you implemented `myPower(2,9)`?
2) Draw call diagrams for both `power(2,9)` and `myPower(2,9)`. Take photos of them and include them in your homework submission as **power_diagram.jpg** and **myPower.jpg.**
3) Which function is more efficient? Why? Try and be as clear as you can.
4) What would have happen if you had implemented the even case of `myPower(b,n)` via:

```
return myPower(b, n/2)*myPower(b, n/2)
```

   Would the solution be correct? Is this alternate version as efficient as the original? Explain.

Problem 5.1c) Empiric Comparison

In this question you will run a computer experiment to verify your conclusions above and to measure the average number of recursive calls these functions make over a range of input values.

First, create the alternate version of the `myPower` that was described in Problem 5.1b Question 4. Call this new version `myOtherPower(b, n)`. Again, Its recursive step for the even case should be:

```
return myOtherPower(b,n/2)*myOtherPower(b,n/2)
```

The recursive step for the odd case should be:

```
return b*myOtherPower(b,n-1)
```

In other words, make sure you do **not** call `power` or `myPower` inside of `myOtherPower`.

Now, modify `myPower` again and modify your new `myOtherPower` function to count the number of recursive calls they make. See the modified version of `power` below as an example. (When you submit, it is okay to include just your final versions of `myPower` and `myOtherPower`.)

```
def power(b, n):
    '''Returns b raised to the nth power'''
    if n < 1:
        return 1
    else:
```

```
        global num_calls
        num_calls = num_calls + 1
        return b * power(b, n - 1)


num_calls = 0
power(2,9)
print(num_calls) # Prints out number of recursive calls made by power
```

Next, modify the additional code provided to find the average number of recursive calls made to each function when they are used to calculate 3^n, for n=1,2,3,4,5,6,...,100.

When your `hw05_1.py` file is run, it should print out these answers. For example:

```
power(2,9) made 9 recursive calls.
myPower(2,9) made __ recursive calls.
myOtherPower(2,9) made __ recursive calls.

The average number of recursive calls when evaluating 3^n, for
n=0,1,2,3,...,100 is as follows:
power = 50.0
myPower = ___
myOtherPower = ___
```

The number of recursive calls for calculating 2^9 reported by your program should agree with your answers in Problem 5.1b, and also be reflected in the number of edges (i.e., lines connecting function calls) in your call diagrams. If they do not agree, something is wrong with your code or your reasoning, or both. Try and resolve it.

~~Part II: Anonymous Functions (20)~~

This problem has been removed from the homework. Move on, be happy!
*As noted on Piazza, We will be giving up to **3 bonus points per question** if you use a lambda in a practical and elegant way anywhere else on the homework!*

Part III: A (Second) Taste of Recursion (20)

Place your answers to this part in `hw05_3.py`.

Recursive functions are functions that may call themselves. Below is an example that calls itself whenever its first argument is less than its second.

```
def myst(a, b):
    """
    Perform a mystery operation on two integers.
    """
    if a >= b:
        return b
    else:
        m = myst(a * 2, b // 2)
        return a + m
```

Trace the execution of `myst(1,32)` and determine what it will return. You may use any reasonable approach, provided that you show all of the calls to the function in a table like the one below. This table is provided in `hw05_3.py`, where you should place your final answer. Complete the table by constructing rows as needed and adding the values for `a`, `b`, and `m` on each recursive call, as well as what each call to `myst` returns. Some initial values of `a` and `b` have been filled in for you:

```
function call | a  |  b |  m | return |
-------------------------------------
myst(1,32)    | 1  | 32 |    |        |
myst(2,16)    | 2  | 16 |    |        |
              |    |    |    |        |
```

Here is a full example for `myst(3,20)`:

```
function call | a  | b  | m  | return |
-------------------------------------
myst(3,20)    | 3  | 20 | 11 | 14     |
myst(6,10)    | 6  | 10 | 5  | 11     |
myst(12,5)    | 12 | 5  |    | 5      |
```

You may also find it helpful to trace this code using Python Tutor. However, as this course progresses you should become comfortable with tracing code by hand (especially since you may be asked to do so on the quiz).

Part IV: Longest Common Subsequence (20)

Place your answers for Part IV in the file named `hw05_4.py`.

For this part of the homework you may use any of the programming techniques that we have covered in class: conditionals, recursion, `filter`, `map`, `reduce` and/or list comprehensions.

The veterinarians of Puppy Land need your help! Tensions are rising between the vets and their canine patients, with puppies suddenly going savage for mysterious causes we need to act fast.

Head Vet Joy suspects it may have something to do with DNA sequences and wants to compare healthy and infected puppies.

Write a function `lcs(s1, s2)` that takes two strings `s1` and `s2` and *returns* the *longest common subsequence* (LCS) that they share. The LCS is a string whose letters appear in both `s1` and `s2`; these letters must appear in the same order in both `s1` and `s2`, but not necessarily consecutively. For example:

```
>>> lcs('gattaca', 'tacgaacta')
'gaaca'
>>> lcs('wow', 'whew')
'ww'
>>> lcs('', 'dog')                # first string is empty
''
>>> lcs('abcdefgh', 'efghabcd')     # tie! 'efgh' would also be fine
'abcd'
>>> lcs('babies', 'puppies')
'ies'
>>> lcs('veterinarians', 'puppies') # tie! 'is' would also be fine
'es'
```

Notes:

- If either `s1` or `s2` is the empty string, the LCS is also the empty string.
- Using the `index` function from Homework 4 may allow you to create an elegant solution.
- If there are ties for the LCS, any one of the ties is acceptable. In the last example above, a return value of `'efgh'` would also have been fine, because both `'abcd'` and `'efgh'` are common subsequences of the two strings, and both of these subsequences have a length of 4.
- Here's one possible strategy for the recursive case (*after* you have checked for base cases):
    - If the first characters in the two strings match, include that character in the LCS, and process the remaining characters of the two strings meaning you should return something to the effect of

      ```
      first character + lcs(s1[1:], s2[1:])
      ```

    - Otherwise, if the first characters don't match, make two recursive calls: one that eliminates the first character in `s1`, and one that eliminates the first character in `s2`. Your code will look something like this

      ```
      result1 = lcs(s1[1:], s2)
      result2 = lcs(s1, s2[1:])
      ```

where you should fill in the blanks in the appropriate way. Return the *better* of these two results.

Part V: Caesar Cipher* (20)

Place your answers for Part III in a file named `hw05_5.py`.

Drs. Griffin and Hersh are running some errands for Dr. Joy's huge upcoming procedure on puppy Lola. Lola seems to be able to understand English and is frightened, so the only way Griffin and Hersh can communicate without scaring Lola is through encrypted messages. To encrypt their messages, they move each letter forward in the alphabet by a certain number of places. When Dr. Joy sent the message "Retrieve the supplies from the hospital", she moved each letter forward 1 place: "Sfusjfwf uif tvqqmjft gspn uif iptqjubm".

Griffin and Hersh are getting tired of encrypting and decrypting these messages by hand, and they can never remember what rotation they were using when they are looking at older messages, so they have hired you to write them the function: `encipher`, which takes in a message (as a string) and a number (how many places to rotate each letter by) and returns the encrypted message.

For this part of the homework you may use any of the programming techniques that we have covered in class: conditionals, recursion, `filter, map, reduce` and/or list comprehensions.

Problem 5.5) Encipher

Write a function `encipher(s, n)` that takes as inputs an arbitrary string `s` and a non-negative integer `n` between 0 and 25, and that returns a new string in which the letters in `s` have been "rotated" by `n` characters forward in the alphabet, wrapping around as needed. For example:

```
>>> encipher('hello', 1)
'ifmmp'
>>> encipher('hello', 2)
'jgnnq'
>>> encipher('hello', 4)
'lipps'
```

Upper-case letters should be "rotated" to upper-case letters, and lower-case letters should be "rotated" to lower-case letters. This also applies even if you need to wrap around the end of the alphabet. For example:

```
>>> encipher('XYZ', 3)
'ABC'
>>> encipher('xyz', 3)
'abc'
```

Non-alphabetic characters should be left unchanged:

```
>>> encipher('#caesar!', 2)
'#ecguct!'
```

Notes:

- You can use the built-in functions `ord` and `chr` to convert from single-character strings to integers and back:

  ```
  >>> ord('a')
  97
  >>> chr(97)
  'a'
  ```

- You can use the following test to determine if a character is between `'a'` and `'z'` in the alphabet:

  ```
  if 'a' <= c <= 'z':
  ```

- A similar test will work for upper-case letters.
- We recommend writing a helper function `rot(c, n)` that rotates a single character `c` forward by `n` spots in the alphabet. For example:

  ```
  >>> rot('a', 1)
  'b'
  >>> print(rot('a', 1))
  b
  >>> rot('y', 2)
  'a'
  >>> rot('A', 3)
  'D'
  >>> rot('Y', 3)
  ```

```
'B'
>>> rot('!', 4)
'!'
```

- Once you have `rot(c, n)`, you can write an `encipher` function.

Once you think you have everything working, here are three more examples to try:

```
>>> encipher('xyza', 1)
'yzab'
>>> encipher('Z A', 2)
'B C'
>>> encipher('Caesar cipher? I prefer Caesar salad.', 25)
'Bzdrzq bhogdq? H oqdedq Bzdrzq rzkzc.'
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.