National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2019/2020

Practice Questions Set 1

The questions are of varying difficulty and are meant to reinforce your fundamentals. Please make sure that you try work on the questions in IDLE, and not just try to write code on paper (or worse, in your head). This is important. **Simply thinking or writing out the solution and concepts involved on paper is not likely to be sufficient to help prepare you for the midterm and practical exams!**

We hope that this set of questions will be helpful for your revision on the material for the first half of the course. Good luck!

Procedural Abstraction

1. **[Basic]** The function `shift_left(num, n)` does the following:

```
shift_left(12345, 0) => 12345
shift_left(12345, 1) => 23451
shift_left(12345, 2) => 34512
...
shift_left(12345, 5) => 12345
shift_left(12345, 6) => 23451
shift_left(12345, 7) => 34512
```

   (a) Write a function (in IDLE) `shift_one_left` that takes in a number as parameter and shifts the number by one to the left, i.e. `shift_one_left(num)` should be equivalent to `shift_left(num,1)`.

   (b) Complete the definition of `shift_left`. Again, run the code in IDLE.

   (c) If you've written an iterative version of `shift_left`, write a recursive version. If you've written a recursive version, write an iterative version named `shift_left_alt`.

   (d) Write a function `shift_right` that does the obvious thing, i.e. instead of shifting a number to the left, it shifts it to the right. Write this function in two ways, both recursively and iteratively.

   Note: All inputs in this question are non-negative integers.

2. **[Basic]** Write a function (in IDLE) `nth_digit` such that it returns the nth digit of a number from the rear:

   ```
   nth_digit(1, 12345) => 5
   nth_digit(3, 12345) => 3
   nth_digit(4, 12345) => 2
   nth_digit(10, 12345) => None
   ```

   Write another function `mth_digit`, which returns the mth digit from the front:

   ```
   mth_digit(1, 12345) => 1
   mth_digit(3, 12345) => 3
   mth_digit(4, 12345) => 4
   mth_digit(10, 12345) => None
   ```

   Note: The input number in this question is an integer.

3. **[Basic]** Write a function (in IDLE), `divisible_by_11(num)`, to return the divisibility of num by 11, i.e. `divisible_by_11(num)` should return `True` is num is divisible by 11, or `False` otherwise.

   Note: A number is divisible by 11 if the difference between the sum of odd digits and the sum of even digits is divisible by 11. Note that 0 is divisible by 11. Do not implement the following function (which defeats the purpose of this question and is certainly not what we're looking for):

   ```
   def divisible_by_11(num):
      return num % 11 == 0
   ```

4. **[Basic]** Write a function (in IDLE) `count_instances(num, seq)` that returns the number of times `num` appears in `seq`, e.g.

   ```
   count_instances(3, (1, 2, 3)) => 1
   count_instances(3, (1, 2, 3, 3, 2, 3)) => 3
   count_instances(5, (1, 2, 3)) => 0
   ```

5. **[Basic]** Write a function (in IDLE) `concat(n, m)` that concatenate two numbers. For example, `concat(12345, 67890)` returns `1234567890`.

6. **[Basic]** Write a function (in IDLE) `replace_digit(n, d, r)` which replaces each occurrence of digit `d` in the number `n` by `r`.

   ```
   replace_digit(31242154125, 1, 0) => 30242054025
   ```

7. **[Basic]** For all of the previous questions (except Question 1), implement the function both recursively and iteratively.

8. **[Intermediate, Lecture Notes]** Implement the counting change example given in the lecture notes – implement `count_chage(amount, kinds_of_coins)` which returns the number of ways to return change for a given amount n. The change can be return using coins worth $1, 50 cents, 20 cents, 10 cents, 5 cents or 1 cent. Now without looking at the lecture notes, write the function `count_change` in IDLE.

   ```
   count_change(5, 5) => 2
   count_change(10, 5) => 4
   count_change(20, 5) => 10
   ```

Order of Growth

9. **[Basic]** What are the time and space complexity of `foo1` and `foo2`? Explain, succinctly, what both functions do. Can you write a function that is better than `foo1` and `foo2`? What will be its time and space complexity?

```
def foo1(n):
    counter = 2
    while counter < n:
        if n % counter == 0:
            return True
        counter = counter + 1
    return False

def foo2(n):
    for i in range(2,(n//2+1)):
        if n % i == 0:
            return True
    return False
```

10. **[Basic]** Give the simplified big O notations for the five expressions below, and arrange them in increasing order of growth.

    (a) O($5n^4 + 2n^2 - n$)

    (b) O($3^n + ln(n)$)

    (c) O($n! + 8n^2$)

    (d) O($n\sqrt{n}$)

    (e) O($log_2 20n$)

11. **[Intermediate, Lecture Notes]** Compute the time and space complexity ie. O(n) order of growth for `move_tower` to solve the Towers of Hanoi problem as discussed in lecture. The code is attached below for your reference. *Hint: Try to compute the complexity for 2 discs, then 3 discs, and try to generalise it for n to n+1.*

```
def move_tower(size, src, dest, aux):
    if size == 1:
        print_move(src, dest)
    else:
        move_tower(size-1, src, aux, dest)
        print_move(src, dest)
        move_tower(size-1, aux, dest, src)

def print_move(src, dest):
    print("move top disk from ", src," to ", dest)
```

Higher-order Functions

12. **[Basic, Concrete Abstraction]** Write a higher-order function `exception_function` which will return a function with exceptions. `exception_function` should take in a function f(x), an integer input and an integer output, and return another function g(x). g(x) should be same to f(x), except that when x is the same as the input integer, the output will be returned.

    For example, in the code below, given that we have a function `usually_sqrt` which returns the square root of the argument. Using `exception_function` we obtain `new_sqrt`, which behaves similarly to `usually_sqrt` except when called `new_sqrt(7)`, where the value of `2` will be returned.

    The test case is given below, and if the program is implemented correctly in IDLE, you will expect the outputs stated.

    ```
    new_sqrt = exception_function(usually_sqrt, 7, 2)
    new_sqrt(9) => 3.0
    new_sqrt(16) => 4.0
    new_sqrt(7) => 2
    ```

    Write a function `usually_double` which doubles its argument. Using the functions `usually_double` and `exception_function`, create a function `new_double`, which returns double the argument except when given the argument `4`, `7`, and `11`, in which it returns the value `0`.

13. **[Basic]** Consider the following function-generating functions:

    ```
    from operator import *

    def make_multiplier(scaling_factor):
        return lambda x : mul(x, scaling_factor)

    def make_exponentiator(exponent):
        return lambda x : pow(x, exponent)
    ```

    `mul` takes two variables `x` , `y` and returns `x * y`. `pow` takes two variables `x` , `y` and returns x^y. Notice that these two functions are quite similar. We could abstract out the commonality into an even more general function `make_generator` such that we could then just write:

    ```
    make_multiplier = make_generator(mul)
    make_exponentiater = make_generator(pow)
    ```

    Write the function `make_generator` in IDLE. If you have seen this question before, try it out without referring to your notes.