National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2020/2021

Solutions for Recitation 2
Recursion, Iteration & Orders of Growth

Instructor notes: Usually I talk about Recursion and Iteration first, then discuss Order of Growth after discussing question 1 and before question 2. It kinda flows better though the worksheet has order of growth first.

Definitions

Theta (Θ) notation:

$$f(n) = \Theta(g(n)) \;\Leftrightarrow\; \exists k_1, k_2, n_0 \;.\; k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n), \text{for } n > n_0$$

Big-O notation:

$$f(n) = O(g(n)) \;\Leftrightarrow\; \exists k, n_0 \;.\; f(n) \leq k \cdot g(n), \text{for } n > n_0$$

Adversarial approach: For you to show that $f(n) = \Theta(g(n))$, you pick k_1, k_2, and n_0, then I (the adversary) try to pick an n which doesn't satisfy $k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$.

Instructor notes: Tell students it is for extra information only and will not be tested. No need to discuss the above.

Implications

Ignore constants. Ignore lower order terms. For a sum, take the larger term. For a product, multiply the two terms. Orders of growth are concerned with how the effort scales up as the size of the problem increases, rather than an exact measure of the cost.

Typical Orders of Growth

- $\Theta(1)$ - Constant growth. A fixed number of simple, non-decomposable operations have constant growth.
- $\Theta(\log n)$ - Logarithmic growth. At each iteration, the problem size is scaled down by a constant amount.
- $\Theta(n)$ - Linear growth. At each iteration, the problem size is decremented by a constant amount.
- $\Theta(n \log n)$ - Nifty growth. Nice recursive solution to normally $\Theta(n^2)$ problem.
- $\Theta(n^2)$ - Quadratic growth. Computing correspondence between a set of n things, or doing something of cost n to all n things both result in quadratic growth.
- $\Theta(2^n)$ - Exponential growth. Really bad. Searching all possibilities usually results in exponential growth.

Instructor notes:I will use a graph to show what it means by how the time/space grows as the input grows.

What's n?

Order of growth is *always* in terms of the size of the problem. Without stating what the problem is, and what is considered primitive (what is being counted as a "unit of work" or "unit of space"), the order of growth doesn't have any meaning.

Instructor notes: Highlight that it is very important to properly define n. Sometimes the input of the function is not n, so it is meaningless to say something like $O(n)$ without actually defining n.

Problems

Instructor notes: Before discussing the problems, briefly recap what is recursion and iteration.

Recursion is defining the sub-problem as itself, and has two components: 1) base case and 2) recurrence relation or recursive step.

Iteration is basically looping, and Python has two means of doing it: `while` and `for`. I like to ask what is the relationship between all code that can be written using `while` and `for`, i.e., can every code written using `for` also be written using `while`? Vice versa? Is one a "subset" of the other? Or partially intersect? I ask them to go think about it themselves.

1. Remember our point-of-sale and order-tracking system from last week? Recall that the joint only sells 4 options for combos: Classic Single Combo (hamburger with one patty), Classic Double With Cheese Combo (2 patties), and Classic Triple with Cheese Combo (3 patties), Avant-Garde Quadruple with Guacamole Combo (4 patties). We shall encode these combos as 1, 2, 3, and 4 respectively. Each meal can be *biggie-sized* to acquire a larger box of fries and drink. A *biggie-sized* combo is represented by 5, 6, 7, and 8 respectively, for combos 1, 2, 3, and 4 respectively.

   In addition, an order is a collection of combos. We'll encode an order as each digit representing a combo. For example, the order 237 represents a Double, Triple, and *biggie-sized* Triple.

   Assume that you have the following functions available:

   - `biggie_size` which when given a regular combo returns a *biggie-sized* version.
   - `unbiggie_size` which when given a *biggie-sized* combo returns a non-*biggie-sized* version.
   - `is_biggie_size` which when given a combo, returns `True` if the combo has been *biggie-sized* and `False` otherwise.
   - `combo_price` which takes a combo and returns the price of the combo.
   - `empty_order` which takes no arguments and returns an empty order which is represented by 0.
   - `add_to_order` which takes an order and a combo and returns a new order which contains the contents of the old order and the new combo. For example, `add_to_order(1,2) -> 12`.

   (a) Write a recursive function called `order_size` which takes an order and returns the number of combos in the order. For example, `order_size(237) -> 3`.

   Note: We assume that the input order is a valid one. The solutions for Recitation 1 already incorporate code that does this testing.

```
def order_size(order):
    if order == 0:
        return 0
    else:
        return 1 + order_size(order // 10)
```

Instructor notes: Before writing any code, explain the components of recursion, namely: 1) base case and 2) recurrence relation. Then explain how you derive these two components for this problem.

(b) Write an iterative version of `order_size`.

```
def order_size(order):
    count = 0;
    while order > 0:
        order = order // 10
        count = count + 1

    return count
```

(c) Write a recursive function called `order_cost` which takes an order and returns the total cost of all the combos.

Instructor notes: I just say this function is very similar to `order_size`, and copy and paste the codes from both recursive and iterative and just modify the same differences. Save time examining this function in detail.

```
def order_cost(order):
    if order == 0:
        return 0.0
    else:
        return combo_price(order % 10) + order_cost(order // 10)
```

(d) Write an iterative version of `order_cost`.

```
def order_cost(order):
    cost = 0.0
    while order > 0:
        cost = cost + combo_price(order % 10)
        order = order // 10

    return cost
```

(e) **Homework:** Write a function called `add_orders` which takes two orders and returns a new order that is the combination of the two. For example, `add_orders (123,234) -> 123234`. Note that the order of the combos in the new order is not important as long as the new order contains the correct combos. `add_orders(123,234) -> 122334` would also be acceptable.

```
def add_orders(order_1, order_2):
    if order_2 == 0:
        return order_1
    else:
        return add_orders(order_1*10 + order_2%10, order_2//10)
```

2. Give order notation for the following:

   (a) $5n^2 + n$

   Answer: $O(n^2)$

   (b) $\sqrt{n} + n$

   Answer: $O(n)$

   (c) $3^n n^2$

   Answer: $O(3^n n^2)$

3.
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

   Running time: $O(n)$. Space: $O(n)$.

   Instructor notes: Draw the call stack and show how time and space is linear. You can use Pythontutor to show the call stack

4. Write an iterative version of `fact`.

   Instructor notes: Explain as how it was explained in lecture

   Answer:

```
def fact(n):
    product = 1

    for i in range(2, n+1):
        product *= i

    return product
```

   Running time: $O(n)$. Space: $O(1)$.

5.
```
def find_e(n):
    if n == 0:
        return 1
    else:
        return 1/fact(n) + find_e(n - 1)
```

   Running time: $O(n^2)$. Space: $O(n)$. (Assume iterative `fact`)

   Instructor notes: Show that time is actually a arithmetic progression: $n + (n-1) + \ldots + 2 + 1 = O(n^2)$ For space, students tend to be confused so explain that space, unlike time, can be recycled or reclaimed. So as the execution collapses back, the stack frames are freed. So what we are interested in for space is the maximum space needed to support the execution, i.e., the largest space used at any point in time by the execution.

6. Assume you have a function `is_divisible(n, x)` which returns `True` if `n` is divisible by `x`. It runs in $O(n)$ time and $O(1)$ space. Write a function `is_prime` which takes a number and returns `True` if it is prime and `False` otherwise.

   Answer:

```
import math

def is_prime(x):
    if x == 1:
        return False
    else:
        for i in range(2, int(math.sqrt(x) + 1)):
            if is_divisible(x, i):
                return False

        return True
```

Running time: $O(x^{\frac{3}{2}})$. Space: $O(1)$.

Instructor notes: First show the solution where the range is until $x/2$, i.e. `for i in range(2, x//2):` because it appears intuitive to most students that the "midpoint" of the search should be half of x. In which case the running time is $O(x)$, no different than before.

Then drop the bomb by revealing that $x/2$ is not the real "midpoint", but it is in fact square root of x. That is because factors comes in pairs and the sqrt is where the factor multiplies by itself and where the pairs are separated.

Then change the ending condition of the loop and show that we now have $O(x^{\frac{3}{2}})$ or $O(x\sqrt{x})$.

Point out the $O(n)$ is the wrong answer since there is no n in this function.

If time permits, demonstrate what happens if `is_divisible(n, x)` runs in $O(1)$ time, or $O(x)$ time, or $O(x)$ space.

Note: `import math` is needed in order to use `sqrt()`. Alternatively, `x**0.5` or `pow(x, 0.5)` can be used without importing `math` but `math.sqrt` is more efficient.

In Python 3, `range()` returns an object that generates the numbers on demand. Hence the space complexity is O(1). However in Python 2, `range()` returns a list and the space complexity will be O(n) instead.

Instructor notes: Python 2 is now end-of-life and support will cease in 2020.

7. **Homework:** Write an iterative version of `find_e`.

```
def find_e(n):
    sum = 0

    for i in range(n+1):
        sum = sum + 1/fact(i)

    return sum
```

Running time: $O(n^2)$. Space: $O(1)$. (Assume iterative `fact`)