ORDERS OF GROWTH AND MIDTERM REVIEW

CS 61A

July 14, 2021

1 Orders of Growth

An order of growth (OOG) characterizes the runtime **efficiency** of a program as its input becomes extremely large. Since we care about rate of growth, we ignore constant coefficients and exclusively consider the fastest growing term. For example, on very large inputs, $2n^2 + 3n - 20$ behaves the same as n^2. Common runtimes, in increasing order of time, are: constant, logarithmic, linear, quadratic, and exponential.

Examples:

Constant time means that no matter the size of the input, the runtime of your program is consistent. In the function `f` below, no matter what you pass in for `n`, the runtime is the same.

```
def f(n):
    return 1 + 2
```

A common example of a linear OOG involves a single for/while loop. In the example below, as `n` gets larger, the amount of time to run the function grows proportionally.

```
def f(n):
    while n > 0:
        print(n)
        n -= 1
```

We can modify this while loop to get an example of logarithmic OOG. Suppose that, instead of subtracting 1 each time, we halve the size of n. For n = 1000, the program would take 10 iterations to terminate (since $2^10 = 1024$). The runtime is propotional to $\log(n)$.

```
def f(n):
    while n > 0:
        print(n)
        n /= 2
```

An example of a quadratic runtime involves nested for loops. If you increment the value of n by only 1, an additional n amount of work is being done, since the inner for loop will run one more time. This means that the runtime is proportional to n^2.

```
def f(n):
    for i in range(n):
        for j in range(n):
            print(i*j)
```

1. What is the order of growth for foo?

   (a)
   ```
   def foo(n):
       for i in range(n):
           print('hello')
   ```

   (b) What's the order of growth of foo if we change range(n):

      i. To range(n/2)?

      ii. To range(n**2 + 5)?

      iii. To range(10000000)?

2. What is the order of growth for belgian_waffle?

   ```
   def belgian_waffle(n):
       total = 0
       while n > 0:
           total += 1
           n = n // 2
       return total
   ```

3. **Fast Exponentiation:** in this problem, we will examine a real-world algorithm used to improve the speed of calculating exponents. You can assume for the purposes of

this problem that multiplication is a constant time operation.

(a) First, determine the runtime efficiency of the naive exponentiation algorithm.

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n - 1)
```

(b) Now, express the runtime of the fast exponentiation algorithm.

```
def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0: # Assume square runs in constant time
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

(c) What about this slightly modified version of `fast_exp`?

```
def fast_exp(b, n):
    for _ in range(50 * n):
        print("Killing time")
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

2 Midterm Review - Environment Diagrams

1. Draw the environment diagram that results from running the following code.

```
def yak(zebra):
    return 20 // zebra
def llama(alpaca):
    zebra = 0
    def yak(zebra):
        return alpaca(zebra)
    return yak

llama(yak)(4)
```

2. Draw the environment diagram that results from running the code.

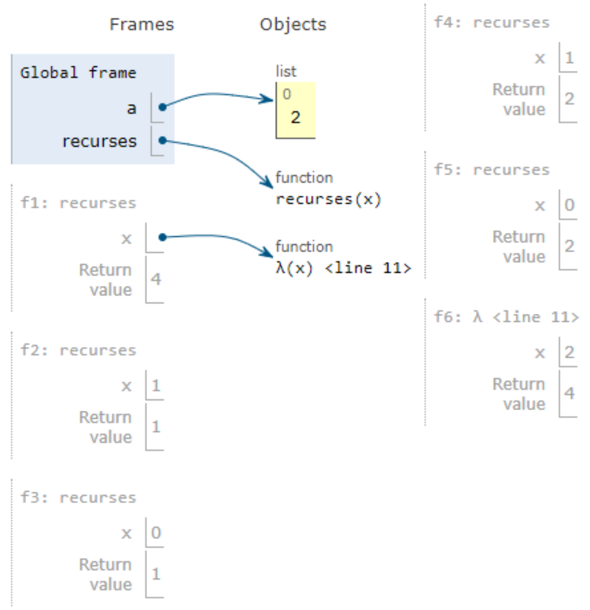
```
def bar(f, x):
    if x == 1:
        return f(x)
    else:
        return f(x) + bar(f, x - 1)

f = 4
bar(lambda x: x + f, 2)
```

3. **recurses**

Fill in each blank in the code below so that its environment diagram is the following. You are not allowed to use operations like +, -, *, /, %, **max**, and **min**.

```
a = [0]
def recurses(x):
    if x == 0:
        return ________________
    elif type(x) == int:
        a[0] += x
        return ________________
    else:
        return _______________\
        ______________________
recurses(lambda x: x * x)
```



3 Midterm Review - Higher Order Functions

1. Make a lambda function, `make_interval()`, that takes in the upper and lower bound of an interval, and returns a function that takes in a value `x` and checks whether `x` is in the interval [`lower`, `upper`], inclusive.

```
>>> make_interval = ______________________________________
>>> in_interval = make_interval(-1, 2)
>>> in_interval(0)
True
>>> in_interval(61)
False
```

2. Implement `make_alternator` which takes in two functions and outputs a function. The returned function takes in a number `x` and prints out all the numbers from 1 to `x`, applying `f` to the odd numbers and applying `g` to the even numbers before printing.

```
def make_alternator(f, g):
    """
    >>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
    >>> a(5)
    1
    6
    9
    8
    25
    """
```

3. Write a function `partial_summer`, which takes in a list of integers `lst` and returns a function. The returned function takes in a non-negative integer `n`. It prints a sum derived from the first `n` elements of `lst`: if element `X` is even, divide `X` by `2` before adding it to the sum, and if `X` is odd. add `1` to `X` before adding it to the sum. If `n > len(lst)`, then sum as many elements of `lst` as you can. After printing the sum, the returned function returns another function, that when called, will perform the same procedure on the remaining `len(lst)- n` elements of `lst`.

```
def partial_summer(lst):
    """
    >>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    >>> f = partial_summer(lst)(3)
    7 # 7 = (1+1) + (2/2) + (3+1)
    >>> g = f(4)
    19 # 19 = (4/2) + (5+1) + (6/2) + (7+1)
    >>> h = g(3)
    14 # 14 = (8/2) + (9+1)
    >>> i = h(1)
    0
    """
    def helper(n):

        total, i = _________, _________

        while ________________ and ____________________:

            if _______________________:

                total += ____________________________
            else:
                total += lst[i] + 1

            __________________________________________
        print(total)

        return _______________________________________
    return helper
```

4 Midterm Review - Recursion

1. Complete the definition for `sum_prime_digits`, which returns the sum of all the prime digits of `n`. Recall that 0 and 1 are not prime numbers. Assume you have access to the function `is_prime` which returns True if a number is prime.

```
def sum_prime_digits(n):
    """
    >>> sum_prime_digits(12345)
    10 # 2 + 3 + 5
    >>> sum_prime_digits(4681029)
    2 # 2 is the only prime number
    """
    if ____________________________________________:

        return ____________________________________

    else:

        ___________________________________________

        if ________________________________________:

            return ________________________________

        return ____________________________________
```

5 Midterm Review - Tree Recursion

1. Imagine you are taking the 61A midterm. There are n questions left on the exam and t time remaining. Completing question i takes i time but gives you $i + 50$ points on the exam. Write a program that will output the highest possible score on the exam given n and t (note: you can't take more than t time).

```
def midterm(n, t):
    """
    >>> midterm(500, 0) # No time left!
    0
    >>> midterm(3, 3) # 51 + 52, questions 1 & 2
    103
    >>> midterm(3, 5) # 52 + 53, questions 2 & 3
    105
    >>> midterm(4, 9) # 52 + 53 + 54, questions 2 & 3 & 4
    159
    """
    if ______________________________________:

        return ______________________________

    else:

        return __________________________
```

2. Write a function that takes as input a number, `n`, and a list of numbers, `lst`, and returns true if we can find a subset of `lst` that sums up to `n`.

```
def add_up(n, lst):
    """
    >>> add_up(10, [1, 2, 3, 4, 5])
    True
    >>> add_up(8, [2, 1, 5, 4, 3])
    True
    >>> add_up(-1, [1, 2, 3, 4, 5])
    False
    >>> add_up(100, [1, 2, 3, 4, 5])
    False
    """
    if ________________________________________:

        return True

    if lst == []:

        ________________________________________________

    else:

        first, rest = ____________________,
            ____________________

        return ________________________________________
```

6 Midterm Review - Lists

1. What would Python display? Draw box-and-pointer diagrams to find out.

   (a)
   ```
   L = [1, 2, 3]
   B = L
   B
   ```

   (b)
   ```
   A = L[1:3]
   L[0] = A
   L = L + A
   B
   ```

2. Write a function `duplicate_list`, which takes in a list of positive integers and returns a new list with each element `x` in the original list duplicated x times.

```
def duplicate_list(lst):
    """
    >>> duplicate_list([1, 2, 3])
    [1, 2, 2, 3, 3, 3]
    >>> duplicate_list([5])
    [5, 5, 5, 5, 5]
    """
    ________________________________

    for ____________________________:

        for ____________________________:

            ____________________________________

    ________________________________
```

7 Midterm Review - Trees

1. Define the function `count` which counts the number of instances of a `value` in the given tree.

```
def count(t, value):
```

2. Write the function `even_square_tree` which takes in a tree `t` and returns a new tree with only the even labels squared.

```
def even_square_tree(t):
    """
    >>> t = tree(2, [tree(1), tree(4)])
    >>> even_square_tree(t)
    tree(4, [tree(1), tree(16)])
    """
    ____________________________________________

    if ________________________________________:

        return ________________________________

    else:

        return ________________________________
```

3. A character tree is a tree where the characters along a path of the tree form a word (as defined in the English dictionary). A path through a tree is a list of adjacent node values that starts from any node and ends with a leaf value.

   Imagine you're playing a version of Scrabble and you really want to win. Implement `scrabble_tree` which takes in a character tree. The function will then find all words in the character tree and return the word with the highest value. You can assume that all characters in the character tree are lower cased.

   You may use the pre-defined functions `is_word(word)` and `score(word)`. The function `is_word(word)` returns True if word is a valid dictionary word and The function `score(word)` returns the score of word in a game of Scrabble. You do not need to worry about how these functions are implemented.

   Note: If all characters have a weight of 1, then this problem is the same as finding the longest string of the character tree.

   (a) First, implement the function `word_exists`, which takes in a word `word` and a character tree `t`. The function will return `True` if characters along a path from the root of `t` to a leaf spells `word`. Otherwise, it returns `False`.

   ```
   def word_exists(word, t):
       if len(word) == 1:
           return ______________________________
       elif ________________:
           return False
       return ____(________________________________)
   ```

   (b) Now, implement the function `scrabble_tree`. You may use the function you defined in part a, as well as the provided functions `is_word(word)` and `score`. You may also want to use the built-in Python function `filter`.

   The function `filter` takes in a single argument function as its first parameter and a sequence as its second parameter. The function will then test which elements of the sequence is `True` using the provided function.

   ```
   >>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   >>> evens = list(filter(lambda x: x % 2 == 0, lst))
   >>> evens
   [2, 4, 6, 8, 10]
   ```

   Note: We have to call list on the output of filter because filter returns an object (which will be covered in a later part of this course).

```
def scrabble_tree(t):
    """
    We assume that all characters have a score of 1.

    >>> t1 = tree('h', [tree('j', [tree('i')])])
    >>> scrabble_tree(t1)
    'hi'
    >>> t2 = tree('i', [tree('l', [tree('l')])])
    >>> t3 = tree('h', [tree('i'), t2])
    >>> scrabble_tree(t3)
    'hill'
    """
    def find_all_words(t):
        if _______________:
            return __________________
        all_words = []
        for b in branches(t):
            words_in_branch = ______________________
            words_from_t = [
               _____________________________________]
            filter_from_t =
               _____________________________________
            all_words =
               _________________________________________
        return __________________
    clean_words = [
       _______________________________________________]
    return max(________________, key=
       __________________________)
```

8 Midterm Review - Abstraction

In the following problem, we will create a **Abstract Data Type (ADT)** to represent a bookshelf object using dictionaries.

In the first section, we will set up the ADT. Here, we will directly work with the internals of the Bookshelf, so don't worry about abstraction barriers for now. Fill in the following functions based on their descriptions (the constructor is given to you):

```
def Bookshelf(capacity):
    """ Creates an empty bookshelf with a certain max
       capacity. """
    return {'size': capacity, 'books': {}}

def add_book(bookshelf, author, title):
    """
    Adds a book to the bookshelf. If the bookshelf is full,
    print "Bookshelf is full!" and do not add the book.
    >>> books = Bookshelf(2)
    >>> add_book(books, 'Jane Austen', 'Pride and Prejudice')
    >>> add_book(books, 'Sheldon Axler', 'Linear Algebra Done
        Right')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    Bookshelf is full!
    """
    if ________________________________:
        print('Bookshelf is full!')
    else:
        if author in bookshelf['books']:
            ____________________________________
        else:
            ____________________________________
```

```
def get_all_authors(bookshelf):
    """
    Returns a list of all authors who have at least one book
       in the bookshelf.
    >>> books = Bookshelf(10)
    >>> add_book(books, 'Jane Austen', 'Pride and Prejudice')
    >>> add_book(books, 'Sheldon Axler', 'Linear Algebra Done
        Right')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    >>> get_all_authors(books)
    ['Jane Austen', 'Sheldon Axler', 'Kurt Vonnegut']
    """
    return ___________________________________
```

```
def get_author_books(bookshelf, author):
    """
    Given an author name, returns a list with
    all books on the bookshelf written by that author.
    >>> books = Bookshelf(100)
    >>> add_book(books, 'Orson Scott Card', "Ender's Game")
    >>> add_book(books, 'Orson Scott Card', 'Speaker for the
       Dead')
    >>> add_book(books, 'J.R.R. Tolkien', 'The Hobbit')
    >>> get_author_books(books, 'Orson Scott Card')
    ['Ender's Game', 'Speaker for the Dead']
    """
    return ________________________________________________
```

Now, complete the function `most_popular_author` **without breaking the abstraction barrier**. In other words, you are not allowed to assume anything about the implementation of a Bookshelf object, or use the fact that it is a dictionary. You can only use the methods above and their stated return values.

```
def most_popular_author(bookshelf):
    """
    Returns the author with the greatest number of books on
       this bookshelf.
    You can assume that the bookshelf is not empty.
    >>> books = Bookshelf(100)
    >>> add_book(books, 'Orson Scott Card', 'Xenocide')
    >>> add_book(books, 'Orson Scott Card', 'Children of the
       Mind')
    >>> add_book(books, 'J.R.R. Tolkien', 'The Hobbit')
    >>> most_popular_author(bookshelf)
    'Orson Scott Card'
    """
    return max(
       ________________________________________________, key=
       ________________________________________________________
       )
```