Discussion 6: Orders of Growth, Midterm Review

This is an online worksheet that you can work on during discussions. Your work is not graded and you do not need to submit anything. The last section of most worksheets is Exam Prep, which will typically only be taught by your TA if you are in an Exam Prep section. You are of course more than welcome to work on Exam Prep problems on your own.

Efficiency

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by **runtime**?

Example 1: `square(1)` requires one primitive operation: multiplication. `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes a *constant* number of operations (1). In other words, this function has a runtime complexity of Θ(1).

As an illustration, check out the table below:

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
...	...	...	...
100	`square(100)`	100*100	1
...	...	...	...
n	`square(n)`	n*n	1

Example 2: `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases **linearly** proportional to the input. In other words, this function has a runtime complexity of Θ(`n`).

As an illustration, check out the table below:

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
...	...	...	...
100	`factorial(100)`	100*99*...*1*1	100
...	...	...	...
n	`factorial(n)`	n*(n-1)*...*1*1	n

Here are some general guidelines for finding the order of growth for the runtime of a function:

- If the function is recursive or iterative, you can subdivide the problem as seen above:
  - Count the number of recursive calls/iterations that will be made in terms of input size `n`.

    - Find how much work is done per recursive call or iteration in terms of input size `n`.
    - The answer is usually the product of the above two, but be sure to pay attention to control flow!
- If the function calls helper functions that are not constant-time, you need to take the runtime of the helper functions into consideration.
- We can ignore constant factors. For example `1000000n` and `n` steps are both linear.
- We can also ignore smaller factors. For example if `h` calls `f` and `g`, and `f` is Quadratic while `g` is linear, then `h` is Quadratic.
- For the purposes of this class, we take a fairly coarse view of efficiency. All the problems we cover in this course can be grouped as one of the following:
    - Constant: the amount of time does not change based on the input size. Rule: `n --> 2n` means `t --> t`.
    - Logarithmic: the amount of time changes based on the logarithm of the input size. Rule: `n --> 2n` means `t --> t + k`.
    - Linear: the amount of time changes with direct proportion to the size of the input. Rule: `n --> 2n` means `t --> 2t`.
    - Quadratic: the amount of time changes based on the square of the input size. Rule: `n --> 2n` means `t --> 4t`.
    - Exponential: the amount of time changes with a power of the input size. Rule: `n --> n + 1` means `t --> 2t`.

Questions

Q1: The First Order...of Growth

What is the efficiency of `rey`?

```
def rey(finn):
    poe = 0
    while finn >= 2:
        poe += finn
        finn = finn / 2
    return
```

What is the efficiency of `mod_7`?

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Midterm Review

This section is **far** longer than a typical discussion, and it is recommended that you also use it as a problem bank for your midterm studies! Best of luck, you got this!!

Reverse Environment Diagrams

Q2: Who - What - When

Fill in the lines below so that the execution of the program would lead to the environment diagram below. You may not use any numbers in any blanks.

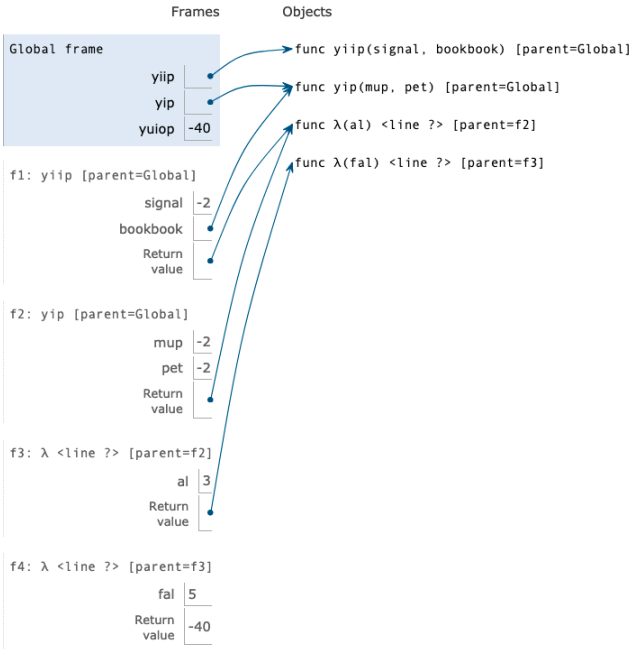
Click here for the diagram that goes along with this problem, go to page 2 of the pdf (https://cs61a.org/assets/pdfs/reverse-ed.pdf)

```
def what(_______):
    def _______(x):
        return ________
    return __________
def who(n):
    def __________(k):
        return 2 * k + n
    return ___________
y = 3
_____(____(____))(4)


```

Q3: Spring 2021: YipYip Book

The following environment diagram was generated by a program:



Click here to open the diagram in a new window (https://i.imgur.com/izi5Wio.png)

In this series of questions, you'll fill in the blanks of the program that follows so that its execution matches the environment diagram. You may want to fill in the blanks in a different order; feel free to answer the questions in whatever order works for you.

```
def yiip(signal, bookbook):
    if signal < 0:
        return ______
               (a)
    elif signal == 0:
        return float("inf")
    return signal * -98

def yip(mup, pet):
    if _______:
        (b)
        mup += 1
    if _______:
        (c)
        return lambda al: ______
                          (d)
    return lambda al: lambda fal: al - fal

yuiop = yiip(_____, _____)(3)(5)
           (e)    (f)
```

Which of these could fill in blank (a)?

i. `lambda al: signal - al * 3` ii. `lambda al: bookbook(signal, 3)(al)` iii. `bookbook` iv. `yiip(bookbook)` v. `yip` vi. `bookbook(-3, signal)` vii. `bookbook(signal + - 3)`

Which of these could fill in blank (b)? **Select all that apply.**

i. `mup < 0` ii. `pet <= 0` iii. `mup == pet` iv. `mup <= 0` v. `pet == 0` vi. `pet < 0`

Which of these could fill in blank (c)? **Select all that apply.**

i. `mup > 0` ii. `mup == -pet` iii. `pet == -mup` iv. `mup == pet` v. `pet > 0` vi. `mup <= 0 and pet <= 0`

Which of these could fill in blank (d)?

i. `al - fal ** fal` ii. `lambda fal: mup ** al + pet ** fal` iii. `mup * al + pet * fal` iv. `lambda fal: mup + pet * fal` v. `mup * pet` vi. `lambda fal: mup * al * pet * fal` vii. `mup ** al + pet ** al` viii. `lambda fal: al * mup + fal * pet`

Which of these could fill in blank (e)?

i. `yiip(2 * -1)` ii. `yiip` iii. `bookbook(yip)` iv. `-2` v. `yip` vi. `signal - 2`

Which of these could fill in blank (f)?

i. `yip` ii. `yip()` iii. `lambda y: y` iv. `yiip` v. `yiip()` vi. `lambda y: yiip(y)` vii. `-2`

Lists and Mutability

Q4: List Comprehension: f

Fill in the definition of `f` below such that the interpreter prints as expected. **Your solution must be on one line.**

```
>>> f = _____________________________________________
>>> f = f(10)
1
2
3
4
5
6
7
8
9
10
```

Then, given your definition of `f`, what will be printed below? (Assuming the above lines have also been executed in the interpreter.)

```
>>> f
```

Q5: Deep map

Write the function `deep_map_mut` that takes a Python list and mutates all of the elements (including elements of sublists) to be the result of calling the function given, `fn`, on each element. Note that the function does not return the mutated list!

> **Hint:** `type(a) == list` will return `True` if `a` is a list.

```
def deep_map_mut(fn, lst):
    """Deeply maps a function over a Python list, replacing each item
    in the original list object.

    Does NOT create new lists by either using literal notation
    ([1, 2, 3]), +, or slicing.

    Does NOT return the mutated list object.

    >>> l = [1, 2, [3, [4], 5], 6]
    >>> deep_map_mut(lambda x: x * x, l)
    >>> l
    [1, 4, [9, [16], 25], 36]
    """
    "*** YOUR CODE HERE ***"


```

HOF and Self Reference

Q6: Foldl

Write a function that takes in a list `s`, a function `f`, and an initial value `start`. This function will fold `s` starting at the beginning. If `s` is `[1, 2, 3, 4, 5]` then the function `f` is applied as follows:

```
f(f(f(f(f(start, 1), 2), 3), 4), 5)
```

You may assume that the function `f` takes in two parameters.

```
from operator import add, sub, mul

def foldl(s, f, start):
    """Return the result of applying the function F to the initial value START
    and the first element in S, and repeatedly applying F to this result and
    the next element in S until we reach the end of the list.

    >>> s = [3, 2, 1]
    >>> foldl(s, sub, 0)      # sub(sub(sub(0, 3), 2), 1)
    -6
    >>> foldl(s, add, 0)      # add(add(add(0, 3), 2), 1)
    6
    >>> foldl(s, mul, 1)      # mul(mul(mul(1, 3), 2), 1)
    6

    >>> foldl([], sub, 100)   # return start if s is empty
    100
    """
    "*** YOUR CODE HERE ***"


```

Q7: Announce Losses

It's Hog again! Write a commentary function `announce_losses` that takes in a player `who` and returns a commentary function that announces whenever that player loses points.

```
def announce_losses(who, last_score=0):
    """
    >>> f = announce_losses(0)
    >>> f1 = f(10, 0)
    >>> f2 = f1(1, 10) # Player 0 loses points due to swine swap
    Oh no! Player 0 just lost 9 point(s).
    >>> f3 = f2(7, 10)
    >>> f4 = f3(7, 11) # Should not announce when player 0's score does not change
    >>> f5 = f4(11, 12)
    """
    assert who == 0 or who == 1, 'The who argument should indicate a player.'
    def say(score0, score1):
        if who == 0:
            score = ____________________________
        elif who == 1:
            score = ____________________________
        if ____________________________:
            ____________________________________________________
        "*** YOUR CODE HERE ***"
        return ____________________________
    return say

```

Recursion

Q8: Pig Latin

Consider the below function `pig_latin`, which computes the pig latin equivalent of an English word

```
def pig_latin_original(w):
    """Return the Pig Latin equivalent of a lowercase English word w."""
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin_original(rest(w) + first(w))

def first(s):
    """Returns the first character of a string."""
    return s[0]

def rest(s):
    """Returns all but the first character of a string."""
    return s[1:]

def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    c = first(w)
    return c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u'
```

This code repeatedly moves a letter from the beginning of a word to the end, until the first letter is a vowel, at which point it adds on `'ay'` to the end. However, this code fails when the original word has no vowels in the set {a, e, i, o, u}, such as the word "sphynx." Write a new version of the `pig_latin` function that just adds `'ay'` to the original word if it does not contain a vowel in this set. Use only the `first`, `rest`, and `starts_with_a_vowel` functions to access the contents of a word, and use the built-in `len` function to determine its length. Do not use any loops.

```
def pig_latin(w):
    """Return the Pig Latin equivalent of a lowercase English word w.

    >>> pig_latin('pun')
    'unpay'
    >>> pig_latin('sphynx')
    'sphynxay'
    """
    "*** YOUR CODE HERE ***"

```

Q9: Ten-pairs

Write a function that takes a positive integer `n` and returns the number of ten-pairs it contains. A ten-pair is a pair of digits within `n` that sums to 10. *Do not use any assignment statements.*

The number 7,823,952 has 3 ten-pairs. The first and fourth digits sum to 7+3=10, the second and third digits sum to 8+2=10, and the second and last digit sum to 8+2=10. Note that a digit can be part of more than one ten-pair.

> *Hint*: Use a helper function to calculate how many times a digit appears in n.

```
def ten_pairs(n):
    """Return the number of ten-pairs within positive integer n.

    >>> ten_pairs(7823952)
    3
    >>> ten_pairs(55055)
    6
    >>> ten_pairs(9641469)
    6
    """
    "*** YOUR CODE HERE ***"


```

Q10: Num Splits

Given a list of numbers `s` and a target difference `d`, write a function `num_splits` that calculates how many different ways are there to split `s` into two subsets, such that the sum of the first is within `d` of the sum of the second. The number of elements in each subset can differ.

You may assume that the elements in `s` are distinct and that `d` is always non-negative.

Note that the order of the elements within each subset does not matter, nor does the order of the subsets themselves. For example, given the list `[1, 2, 3]`, you should not count `[1, 2]`, `[3]` and `[3]`, `[1, 2]` as distinct splits.

> Hint: If the number you return is too large, you may be double-counting somewhere. If the result you return is off by some constant factor, it will likely be easiest to simply divide/subtract away that factor.

```
def num_splits(s, d):
    """Return the number of ways in which s can be partitioned into two
    sublists that have sums within d of each other.

    >>> num_splits([1, 5, 4], 0)  # splits to [1, 4] and [5]
    1
    >>> num_splits([6, 1, 3], 1)  # no split possible
    0
    >>> num_splits([-2, 1, 3], 2) # [-2, 3], [1] and [-2, 1, 3], []
    2
    >>> num_splits([1, 4, 6, 8, 2, 9, 5], 3)
    12
    """
    "*** YOUR CODE HERE ***"


```

Trees

Q11: Pruning Leaves

Define a function `prune_leaves` that given a tree `t` and a tuple of values `vals`, produces a version of `t` with all its leaves that are in `vals` removed. Do not attempt to try to remove non-leaf nodes and do not remove leaves that do not match any of the items in `vals`. Return `None` if pruning the tree results in there being no nodes left in the tree.

```
def prune_leaves(t, vals):
    """Return a modified copy of t with all leaves that have a label
    that appears in vals removed.  Return None if the entire tree is
    pruned away.

    >>> t = tree(2)
    >>> print(prune_leaves(t, (1, 2)))
    None
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> print_tree(numbers)
    1
      2
      3
        4
        5
      6
        7
    >>> print_tree(prune_leaves(numbers, (3, 4, 6, 7)))
    1
      2
      3
        5
      6
    """
    "*** YOUR CODE HERE ***"

```

Q12: Hailstone Tree

We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number `n`, continuing to `n/2` if `n` is even or `3n+1` if `n` is odd, ending with 1. Write a function `hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.

> **Hint:** A node of a hailstone tree will always have at least one, and at most two branches (which are also hailstone trees). Under what conditions do you add the second branch?

```
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will reach N, with height H.
    >>> print_tree(hailstone_tree(1, 0))
    1
    >>> print_tree(hailstone_tree(1, 4))
    1
        2
            4
                8
                    16
    >>> print_tree(hailstone_tree(8, 3))
    8
        16
            32
                64
            5
                10
    """
    if ______________________________________:
        return ______________________________________
    branches = ______________________________________
    if ____________ and ____________ and ____________:
        branches += ______________________________________
    return tree(n, branches)

def print_tree(t):
    def helper(i, t):
        print("    " * i + str(label(t)))
        for b in branches(t):
            helper(i + 1, b)
    helper(0, t)
```