CS 61A Structure and Interpretation of Computer Programs
Summer 2023

Midterm Solutions

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

You can complete and submit these questions before the exam starts.

(a) What is your full name?

Oski the Bear

(b) What is your student ID number?

123456789

(c) What is your @berkeley.edu email address?

oski@berkeley.edu

(d) Sign (or type) your name to confirm that all work on this exam will be your own. The penalty for academic misconduct on an exam is an F in the course.

Oski Bear

1. (8.0 points) What Would Python Display

For each expression below, choose the correct option or write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines.

- If an error occurs, write "Error", but include all output displayed before the error.
- If evaluation would run forever, write "Forever".
- To display a function value, write "Function".
- If the evaluated expression wouldn't display anything, write "Nothing". The interactive interpreter displays the value of a successfully evaluated expression, unless it is None.

(a) i. (0.5 pt)

```
>>> (1 or 2) and (3 or 4)
```

- ○ `True`
- ○ `False`
- ○ `0`
- ○ `1`
- ○ `2`
- ● `3`
- ○ `4`

ii. (0.5 pt)

```
>>> ('Barbie' or 'Ken') * 2
```

- ○ `'Barbie'`
- ● `'BarbieBarbie'`
- ○ `'Ken'`
- ○ `'KenKen'`
- ○ `'BarbieKen'`
- ○ `'KenBarbie'`
- ○ `'BarbieBarbieKenKen'`
- ○ `2`
- ○ `Error`
- ○ `Nothing`

iii. (0.5 pt) For the rest of this question, assume the following code has been executed.

```
>>> it = iter([2, ['a', 'b'], 'c', 3])
>>> next(it)
```

```
2
```

iv. (0.5 pt)

```
>>> next(it)
```

```
['a', 'b']
```

v. (1.0 pt)

```
>>> list(it)
```

```
['c', 3]
```

vi. (1.0 pt)

```
>>> next(it)
```

```
Error
```

vii. (1.0 pt) For the rest of this question, assume the following code has been executed.

```
dreamhouse = 101
beach = lambda pink: pink(dreamhouse)
dreamhouse = 102

def is_even(n):
    return n % 2 == 0
```

```
>>> beach(is_even)
```

- (●) True
- ( ) False
- ( ) 101
- ( ) 102
- ( ) 1
- ( ) 0
- ( ) Function
- ( ) Error
- ( ) Nothing

viii. (1.0 pt)

```
>>> beach(dreamhouse)
```

- ( ) True
- ( ) False
- ( ) 101
- ( ) 1
- ( ) 0
- ( ) Function
- (●) Error
- ( ) Nothing

ix. (1.0 pt)

```
>>> pink = beach(print)
```

- ◯ `True`
- ◯ `False`
- ◯ `101`
- ● `102`
- ◯ `1`
- ◯ `0`
- ◯ `Function`
- ◯ `Error`
- ◯ `Nothing`

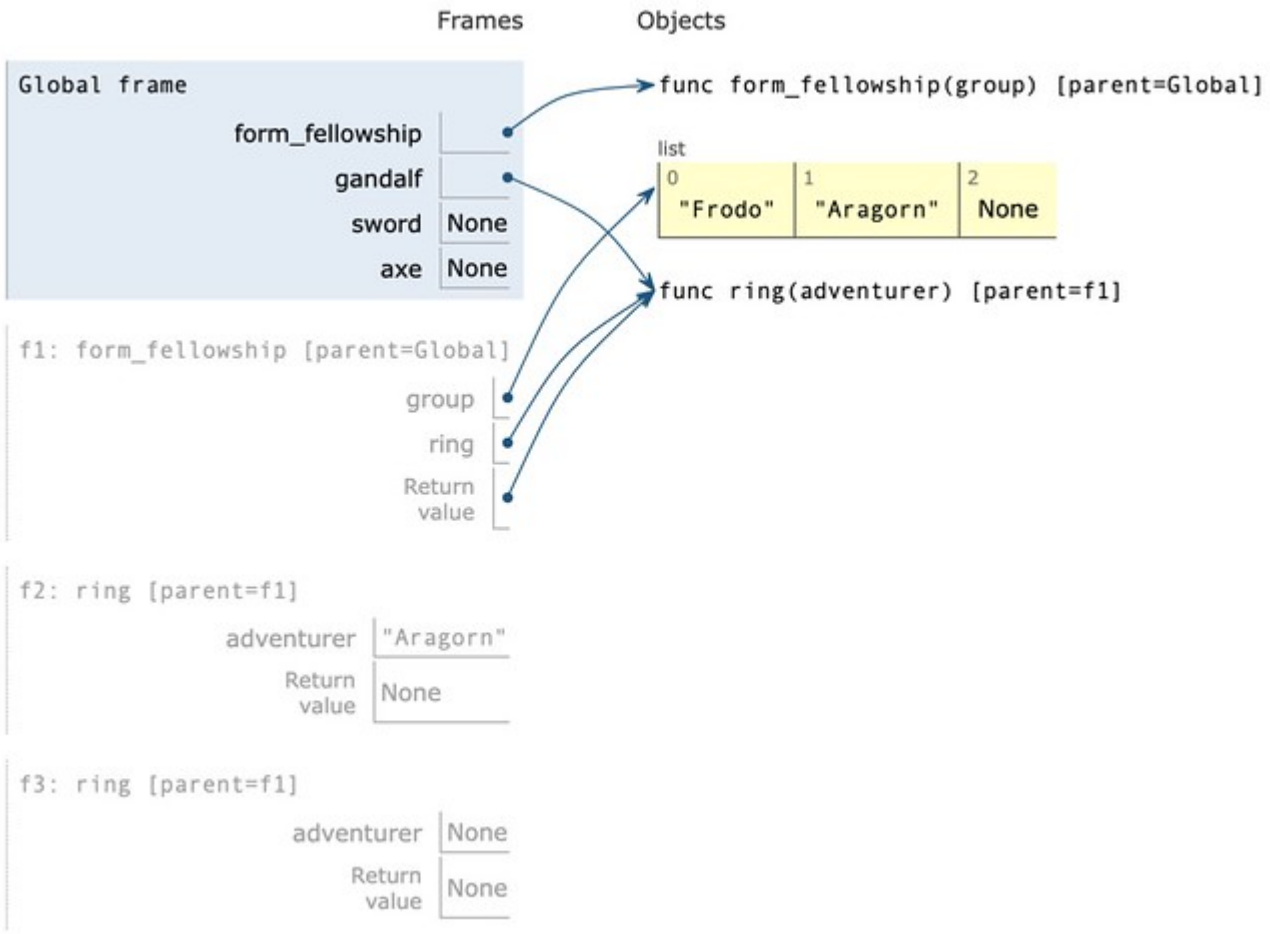
x. (1.0 pt)

```
>>> pink
```

- ◯ `True`
- ◯ `False`
- ◯ `101`
- ◯ `102`
- ◯ `1`
- ◯ `0`
- ◯ `Function`
- ◯ `Error`
- ● `Nothing`

2. (10.0 points) The Fellowship of the List

Given the environment diagram and skeleton code, answer the questions below.




```
def form_fellowship(__1__):
    def ring(__2__):
        return ____3____
    return ____4____

gandalf = form_fellowship(____5____)
sword = _____6_____
_____7_____ = _____8_____
```

(a) i. (1.0 pt) Fill in blank (1)

```
group
```

ii. (1.0 pt) Fill in blank (2)

```
adventurer
```

iii. **(1.5 pt)** Which of these could go in 3? (select all that apply)

- ☐ `group + adventurer`
- ☐ `adventurer + group`
- ☐ `group.extend(adventurer)`
- ■ `group.extend([adventurer])`
- ☐ `adventurer.extend(group)`
- ☐ `adventurer.extend([group])`
- ■ `group.append(adventurer)`
- ☐ `group.append([adventurer])`
- ☐ `adventurer.append(group)`
- ☐ `adventurer.append([group])`
- ☐ `group + adventurer[:]`
- ☐ `group[:] + adventurer`

iv. **(1.0 pt)** Fill in blank (4)

```
ring
```

v. **(1.5 pt)** Which of these could go in 5? (select one)

- ● `['Frodo']`
- ○ `'Frodo'`
- ○ `['Frodo', 'Aragorn']`
- ○ `'Aragorn'`
- ○ `['Frodo', 'Aragorn', None]`
- ○ `None`

vi. **(2.0 pt)** Fill in blank (6). You may only make one call expression and may not use `or`, `and`, `if`, or `else`.

```
gandalf("Aragorn")
```

vii. **(1.0 pt)** Fill in blank (7)

```
axe
```

viii. (1.0 pt) Which of these could go in 8? (select all that apply)

- [x] `gandalf(sword)`
- [ ] `gandalf(sword.append(None))`
- [ ] `gandalf(group)`
- [ ] `gandalf(adventurer)`
- [x] `gandalf(None)`
- [ ] `sword('Aragorn')`
- [ ] `sword(None)`

3. (13.0 points) Digit Widgets

In this question, we will be testing your understanding of both iterative and recursive solutions.

Hint: `pow(b, n)` raises `b` to the power of `n`. For example, `pow(10, 3)` is `1000`.

(a) (5.0 points)

Implement the recursive version of `digit_widget` which takes digit `d`. It returns a function with positive integer parameter `n` that removes all instances of `d` from `n`. **You may not use `str` or `repr` or `[` or `]` or `for`.**

```
def digit_widget(d):
    """
    Given a digit D, returns a function which accepts positive integer N that removes
    all instances of D from N. If there are no digits remaining in N, return 0.
    >>> remove_5s = digit_widget(5)
    >>> remove_5s(1234512345)
    12341234
    >>> remove_5s(55555)
    0
    >>> remove_0s = digit_widget(0)
    >>> remove_0s(102001)
    121
    >>> remove_0s(900)
    9
    >>> remove_0s(0)
    0
    """
    def remove_digit(n):
        if _________:
              (a)
            return _________
                       (b)
        if n % 10 == _________:
                         (c)
            return _________
                       (d)
        return _________
                   (e)
    return _________
              (f)
```

i. (0.5 pt) Fill in blank (a).

- ◯ `n == d`
- ● `n == 0`
- ◯ `n % 10 == 0`
- ◯ `n > 0`
- ◯ `n >= 0`
- ◯ `n < 0`

ii. **(0.5 pt)** Fill in blank (b)

- (●) `0`
- ( ) `1`
- ( ) `d`
- ( ) `n - 1`
- ( ) `pow(10, d)`
- ( ) `pow(10, n)`

iii. **(0.5 pt)** Fill in blank (c)

- ( ) `0`
- ( ) `1`
- (●) `d`
- ( ) `n - 1`
- ( ) `pow(10, d)`
- ( ) `pow(10, n)`

iv. **(1.0 pt)** Fill in blank (d)

- ( ) `remove_digit(n % d)`
- (●) `remove_digit(n // 10)`
- ( ) `remove_digit(n % 10)`
- ( ) `remove_digit(n - 1)`
- ( ) `remove_digit(n % pow(10, d))`
- ( ) `remove_digit(n // pow(10, d))`

v. **(2.0 pt)** Fill in blank (e).

```
remove_digit(n//10) * 10 + n % 10
```

vi. **(0.5 pt)** Fill in blank (f).

```
remove_digit
```

(b) (5.0 points)

Implement the iterative version of `digit_widget`, `digit_widget_iter`, which takes digit `d`. It returns a function with positive integer parameter `n` that removes all instances of `d` from `n`. **You may not use `str` or `repr` or `[` or `]` or `for`.**

```
def digit_widget_iter(d):
    """
    Given a digit D, returns a function which accepts positive integer N that removes
    all instances of D from N. If there are no digits remaining in N, return 0.
    >>> remove_5s = digit_widget_iter(5)
    >>> remove_5s(1234512345)
    12341234
    >>> remove_5s(55555)
    0
    >>> remove_0s = digit_widget_iter(0)
    >>> remove_0s(102001)
    121
    >>> remove_0s(900)
    9
    >>> remove_0s(0)
    0
    """
    def remove_digit(n):
        result, i = 0, 0
        while _________:
                  (a)
            if _________:
                   (b)
                result += _________
                               (c)

                _________
                   (d)

            _________
               (e)
        return result
    return _________
              (f)
```

i. (0.5 pt) Fill in blank (a).

- ◯ `n == d`
- ◯ `n == 0`
- ◯ `n % 10 == 0`
- ● `n > 0`
- ◯ `n >= 0`
- ◯ `n < 0`

ii. **(0.5 pt)** Fill in blank (b)

- ◯ `n % 10 == 0`
- ◯ `n // 10 != 0`
- ◯ `n % 10 == d`
- ● `n % 10 != d`
- ◯ `n % pow(10, d) == 0`
- ◯ `n // 10 == pow(10, d)`

iii. **(2.0 pt)** Fill in blank (c)

```
pow(10, i) * n % 10
```

iv. **(0.5 pt)** Fill in blank (d).

- ◯ `n = n % pow(10, i)`
- ◯ `i = pow(10, i)`
- ● `i += 1`
- ◯ `n = n % 10`
- ◯ `n = n // 10`
- ◯ `n -= 1`

v. **(1.0 pt)** Fill in blank (e).

- ◯ `n = n % pow(10, i)`
- ◯ `i = pow(10, i)`
- ◯ `i += 1`
- ◯ `n = n % 10`
- ● `n = n // 10`
- ◯ `n -= 1`

vi. **(0.5 pt)** Fill in blank (f).

```
remove_digit
```

(c) (3.0 points)

Implement `digit_machine`, which takes a positive integer `n`. It returns `n` where all instances of 4 and 8 have been removed. **You may not use `str` or `repr` or `[` or `]` or `for`.** Assume `digit_widget` is implemented correctly (and you may use it).

```
def digit_machine(n):
    """
    Given an integer N, return a modified N such that all instances of 4 and 8 have been removed.
    >>> digit_machine(484848)
    0
    >>> digit_machine(123456789)
    1235679
    >>> digit_machine(208)
    20
    """
    return _________
              (a)
```

i. (3.0 pt) Fill in blank (a).

```
return digit_widget(4)(digit_widget(8)(n))
```

4. (8.0 points) Goatda

This question will test the debugging skills you've learned in the class so far. We have included an implementation of the `goatda` function which accepts a single argument function `lamb` and a non-negative integer `n`. It returns a single argument function that applies `lamb` to the argument `n` times. However, **the included implementation is buggy**!

```
def goatda(lamb, n):
    """
    Implement goatda which accepts LAMB, a single argument function, and N the number of times
    to apply LAMB. Return a single argument function that applies LAMB to the argument N times.
    >>> add3 = goatda(lambda x: x+1, 3)
    >>> add3(10) # (((10 + 1) + 1) + 1)
    13
    >>> add3(2) # (((2 + 1) + 1) + 1)
    5
    >>> print2 = goatda(print, 2)
    >>> print2("hi") # print(print("hi"))
    hi
    None
    >>> identity = goatda(lambda x: x+3, 0)  # The function is applied 0 times
    >>> identity(0)
    0
    """
    if n == 0:                                # line 1
        return lambda x: x                    # line 2
    return lambda x: lamb(goatda(lamb, n-1)) # line 3
```

(a) After implementing `goatda`, we execute the following lines of code in the terminal.

```
>>> add3 = goatda(lambda x: x+1, 3)
>>> add3(10)
```

The call to `add3(10)` results in an error!

i. (2.0 pt) What kind of error does this result in?

- ○ `RecursionError: too much recursion`
- ● `TypeError: unsupported operand type(s)`
- ○ `TypeError: 'NoneType' object is not callable`
- ○ `IndentationError`
- ○ `IndexError`
- ○ `NameError`

ii. (2.0 pt) Let's modify the code, so that it no longer errors. Select how to replace the 3 lines, so that the function passes the doctests and no longer errors. Select `No change` if you do not want to modify the line. Indentation will remain the same.

How would you modify line 1?

- ○ `if n > 0:`
- ○ `if n < 0:`
- ○ `if n >= 0:`
- ○ `if n % 10 > 0:`
- ○ `if n % 10 >= 0:`
- ○ `if lamb(n) == 0:`
- ○ `if lamb(n):`
- ● `No change`

iii. (2.0 pt) How would you modify line 2?

- ○ `return lamb(n)`
- ○ `return n`
- ○ `return lambda x: n`
- ○ `return lambda x: 0`
- ○ `return lambda x: lamb(x)`
- ○ `return lambda x: lambda y: lamb(y)`
- ● `No change`

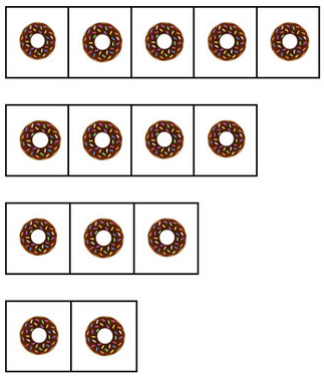
iv. (2.0 pt) How would you modify line 3?

- ○ `return goatda(lamb, lamb(n-1))`
- ○ `return lamb(goatda(lamb, n-1))`
- ○ `return lambda x: lamb(x) + goatda(lamb, n-1)`
- ○ `return lambda x: goatda(lamb(x), n-1)`
- ● `return lambda x: lamb(goatda(lamb, n-1)(x))`
- ○ `return lambda x: goatda(lamb, n-1)(n)`
- ○ `No change`

5. (13.0 points) Sweetness Overload

(a) (3.0 points)

Implement `donut_tower` which takes in the number of donuts `n` and positive integer `k`. It returns the number of layers one can fill using `n` donuts, given the smallest layer consists of `k` donuts and each subsequent layers consists of one more donut than the previous one. For instance, if `n` is 16 and `k` is 2, then you can create 4 layers (as shown below) using 14 donuts.



```
def donut_tower(n, k):
    """
    Given non-zero numbers N and K, returns the number of donut layers
    that can be formed using N donuts and K-donut smallest layer
    >>> donut_tower(0, 3)
    0
    >>> donut_tower(1, 1)
    1
    >>> donut_tower(10, 2) # 2 + 3 + 4 = 9 donuts
    3
    >>> donut_tower(20, 3)  # 3 + 4 + 5 + 6 = 18 donuts
    4
    """
    if _________:
          (a)
       return _________
                 (b)
    else:
       return 1 + _________
                    (c)
```

i. (1.0 pt) Fill in blank (a)

- (●) `n < k`
- ( ) `n >= k`
- ( ) `n > k`
- ( ) `n == 1`
- ( ) `n == 0`
- ( ) `n < 1`

ii. **(1.0 pt)** Fill in blank (b)

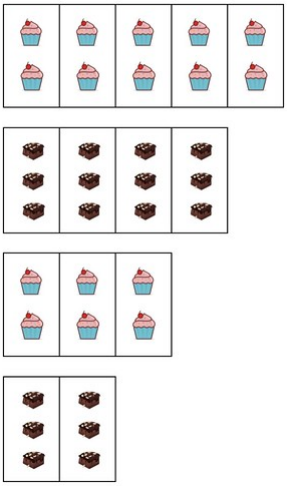
```
0
```

iii. **(1.0 pt)** Fill in blank (c)

- ◯ `max(donut_tower(n, k + 1), donut_tower(n - k, k))`
- ◯ `donut_tower(n, k + 1)`
- ● `donut_tower(n - k, k + 1)`
- ◯ `donut_tower(n - k, k)`
- ◯ `donut_tower(n - k + 1, k)`
- ◯ `donut_tower(n - 1, k)`

(b) (4.0 points)

Implement `alternating_tower`, which takes a positive integer `c`, the number of cupcakes, positive integer `b`, the number of brownies, and `k`, the size of the smallest layer that can hold `k` donuts. It returns the maximum numbers of dessert layers that can be formed using brownies and cupcakes. Since those items are smaller than donuts, each k-sized donut layer can hold twice as many cupcakes and thrice as many brownies. Also, a dessert tower has alternating layers of brownies and cupcakes, but **the first layer must always consist of brownies.** For instance, if `b` is 20, `c` is 20 and `k` is 2, then you can create a maximum of 4 layers (as shown below) using 18 brownies and 16 cupcakes.



```
def alternating_tower(b, c, k):
    """
    >>> alternating_tower(5, 4, 1) # 1st layer has 3 brownies, 2nd layer has 4 cupcakes
    2
    >>> alternating_tower(10, 15, 3) # 1st layer has 9 brownies, 2nd layer has 8 cupcakes
    2
    >>> alternating_tower(16, 6, 5) # 1st layer has 15 brownies
    1
    """
    def helper(b_item, c_item, curr_layer, is_cupcakes):
        if _________:
              (a)
            return 0
        elif is_cupcakes:
            return _________
                      (b)
        else:
            return 1 + helper(_________, c_item, curr_layer + 1, not is_cupcakes)
                                    (c)
    return helper(_________, c, k + 1, True)
                     (d)
```

i. **(1.0 pt)** Fill in blank (a)

- (●) `b_item < 0 or c_item < 0`
- ( ) `b_item`
- ( ) `b_item == 0 or c_item == 0`
- ( ) `b_item > 0 or c_item > 0`
- ( ) `c_item`
- ( ) `b_item < 0 and c_item < 0`
- ( ) `c_item * 2 == k or b_item * 3 == k`
- ( ) `b_item * 3 <= k`

ii. **(1.0 pt)** Fill in blank (b)

- ( ) `helper(b_item - 3 * curr_layer, c_item, curr_layer + 1, is_cupcakes) - 1`
- ( ) `1 + helper(b_item - 3 * curr_layer, c_item - 2 * curr_layer, curr_layer - 1, not is_cupcake`
- (●) `1 + helper(b_item, c_item - 2 * curr_layer, curr_layer + 1, not is_cupcakes)`
- ( ) `helper(b_item - 3 * curr_layer, c_item - 2 * curr_layer, curr_layer - 1, is_cupcakes)`
- ( ) `helper(b_item, b_item - 2 * curr_layer, curr_layer + 1, not is_cupcakes)`
- ( ) `helper(b_item - 3 * curr_layer, c_item, curr_layer + 1, not is_cupcakes) + helper(b_item, c_item - 2 * curr_layer, curr_layer + 1, is_cupcakes)`

iii. **(1.0 pt)** Fill in blank (c)

- ( ) `b_item`
- ( ) `b_item - 3 * (curr_layer + 1)`
- (●) `b_item - 3 * curr_layer`
- ( ) `3 * b_item`
- ( ) `b_item + c_item * 3 * curr_layer`
- ( ) `b_item // 3 * (curr_layer + 1)`
- ( ) `b_item // 3 * curr_layer`
- ( ) `b_item - 3 // curr_layer`

iv. **(1.0 pt)** Fill in blank (d)

```
b - 3 * k
```

(c) (6.0 points)

Implement `sweetest_tower`, which takes a positive integer `c`, the number of cupcakes, positive integer `b`, the number of brownies, and `k`, the size of the smallest layer that can hold `k` donuts. It returns the maximum number of dessert layers that can be formed using brownies and cupcakes. As previously stated, since those items are smaller than donuts, each k-sized donut layer can hold twice as many cupcakes and thrice as many brownies. **Unlike `alternating_tower`, layers can be in any order.**

```
def sweetest_tower(b, c, k):
    """
    >>> sweetest_tower(0, 2, 1) # 1st layer with 2 cupcakes
    1
    >>> sweetest_tower(5, 4, 4)
    0
    >>> sweetest_tower(17, 19, 3) # layer 1 has 6 cupcakes, 2nd has 8 cupcakes, 3rd has 15 brownie
    3
    >>> sweetest_tower(30, 8, 4)    # 1st layer has 12 brownies, 2nd has 15 brownies
    2
    """
    with_brownies, with_cupcakes = 0,0
    if _________:
          (a)
        with_brownies = _________
                            (b)
    if _________:
          (c)
        with_cupcakes = _________
                            (d)
    return _________(with_brownies, with_cupcakes)
              (e)
```

i. (1.0 pt) Fill in blank (a)

- (●) `b >= 3 * k`
- ( ) `b > 3 * k`
- ( ) `b`
- ( ) `b == 3 * k`
- ( ) `b > c`
- ( ) `b // 3 < k`
- ( ) `b // 3 == k`
- ( ) `b >= 0`

ii. **(1.0 pt)** Fill in blank (b)

- ○ `1 + sweetest_tower(b // 3 - k, c, k - 1)`
- ○ `sweetest_tower(b - 3 * k, c, k + 1)`
- ● `1 + sweetest_tower(b - 3 * k, c, k + 1)`
- ○ `sweetest_tower(b - 3 * k, c, k - 1)`
- ○ `3 * sweetest_tower(b, c - 2 * k, k + 1)`
- ○ `3 + sweetest_tower(b, c - 2 * k, k + 1)`
- ○ `sweetest_tower(b - 3 // k, c - 2 * k, k + 1)`

iii. **(1.0 pt)** Fill in blank (c)

```
c >= 2 * k
```

iv. **(1.0 pt)** Fill in blank (d)

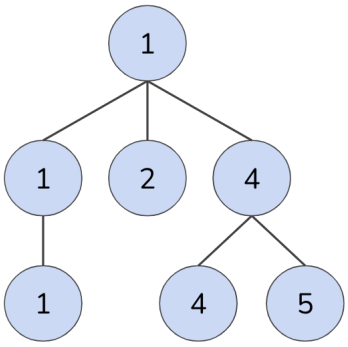
```
1 + sweetest_tower(b, c - 2*k, k + 1)
```

v. **(1.0 pt)** Fill in blank (e)

```
max
```

6. (12.0 points) All Treeils Lead to Rome

Definition: A **Treeil** is a tree where *exactly* one of the child node(s) of each non-leaf node has a label equal to the label of the node. Alternatively, a treeil is a tree where every node has a singular path from the node to a leaf consisting entirely of nodes with the same label. Here is an example of a valid treeil:



(a) (7.0 points) Treeil Trial

Implement `is_treeil` which takes in a tree `t` and determines whether `t` is a treeil.

```
def is_treeil(t):
    """
    Returns whether a tree is a treeil. A treeil is a tree where exactly one of
    the branches of each non-leaf node has a label equal to the label of the node.
    >>> t1 = tree(1)
    >>> is_treeil(t1)
    True
    >>> t2 = tree(1, [tree(2), tree(3)])
    >>> is_treeil(t2)
    False
    >>> t3 = tree(1, [tree(1, [tree(1), tree(3)]), tree(2)])
    >>> is_treeil(t3)
    True
    >>> t4 = tree(1, [tree(1, [tree(1), tree(1)]), tree(2)])
    >>> is_treeil(t4)
    False
    >>> t5 = tree(2, [tree(3, [tree(1)]), tree(2, [tree(2)])])
    >>> is_treeil(t5)
    False
    """
    if _________:
          (a)
        return True
    else:
        match_one = _________([b for b in branches(t) if _________]) == 1
                       (b)                                  (c)
        return _________ and _________([_________ for b in branches(t)])
                  (d)           (e)        (f)
```

i. **(1.0 pt)** Fill in blank (a). Select **all** that apply.

- ☐ `True`
- ☐ `False`
- ■ `is_leaf(t)`
- ☐ `branches(t)`
- ■ `not branches(t)`
- ☐ `label(t)`
- ☐ `not label(t)`
- ■ `branches(t) == []`

ii. **(1.0 pt)** Fill in blank (b).

- ○ `sum`
- ○ `max`
- ○ `min`
- ● `len`
- ○ `any`
- ○ `all`

iii. **(2.0 pt)** Fill in blank (c).

```
label(t) == label(b)
```

iv. **(1.0 pt)** Fill in blank (d).

```
match_one
```

v. **(1.0 pt)** Fill in blank (e).

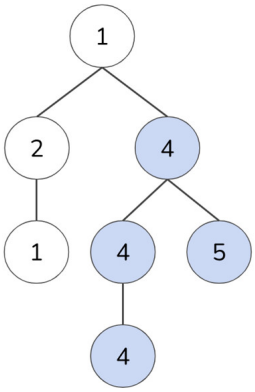
- ○ `sum`
- ○ `max`
- ○ `min`
- ○ `len`
- ○ `any`
- ● `all`

vi. **(1.0 pt)** Fill in blank (f).

```
is_treeil(b)
```

(b) (5.0 points) Treeil Trail

Implement `max_treeil` which takes in a tree `t` and determines the highest number of nodes in a treeil that is a subtree of `t`. You may assume `is_treeil` is implemented correctly from the previous question. The following diagram displays a tree `t`, which is not a treeil. The treeil subtree of `t` with the highest number of nodes is highlighted. Calling `max_treeil` on this tree `t` should give a result of `4`.




```
def max_treeil(t):
    """
    Given a tree (that is not necessarily a treeil), returns
    the highest number of nodes in a treeil in the tree.
    >>> t1 = tree(1)
    >>> max_treeil(t1)
    1
    >>> t2 = tree(1, [tree(2), tree(3)])
    >>> max_treeil(t2)
    1
    >>> t3 = tree(1, [tree(1, [tree(1), tree(3)]), tree(2)])
    >>> max_treeil(t3)
    5
    >>> t4 = tree(1, [tree(1, [tree(1), tree(1)]), tree(2)])
    >>> max_treeil(t4)
    1
    >>> t5 = tree(2, [tree(3, [tree(1)]), tree(2, [tree(2)])])
    >>> max_treeil(t5)
    2
    """
    if _________:
          (a)
        return 1
    elif _________:
            (b)
        return 1 + _________
                      (c)
    else:
        return max([_________ for b in branches(t)])
                       (d)
```

i. **(1.0 pt)** Fill in blank (a).

- ○ `is_treeil(t)`
- ○ `not is_treeil(t)`
- ● `is_leaf(t)`
- ○ `is_treeil(t) or is_leaf(t)`
- ○ `not is_treeil(t) and is_leaf(t)`

ii. **(1.5 pt)** Fill in blank (b).

```
is_treeil(t)
```

iii. **(1.0 pt)** Fill in blank (c).

- ○ `max([max_treeil(b) for b in branches(t)])`
- ○ `len([max_treeil(b) for b in branches(t)])`
- ● `sum([max_treeil(b) for b in branches(t)])`
- ○ `sum([b for b in branches(t) if is_treeil(b)])`
- ○ `len([b for b in branches(t) if is_treeil(b)])`

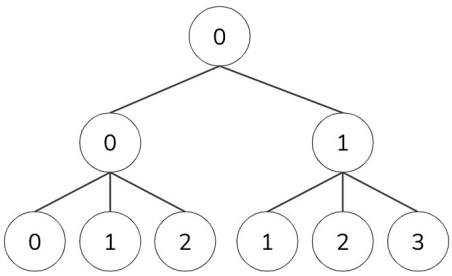
iv. **(1.5 pt)** Fill in blank (d).

```
max_treeil(b)
```

(c) (0.0 points) Treeil Treeatment (A+ Question)

This A+ question is not worth any points. This can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

Implement `make_treeil` which takes in a non-negative integer `n` and returns a treeil of depth `n`. Each node of the treeil has exactly one more child than its parent node. The first branch of a node will have value equal to its parent. Subsequent branches will have a label of exactly one more than the previous branch. The root has a label of 0 and has exactly two children for a tree of depth 1. Here's an example of a treeil of depth 2.




```
def make_treeil(n):
    """
    Returns a treeil of depth n where each node has one more child than its parent.
    The first branch of a node will have label equal to its parent with subsequent
    branches having label of exactly one more than the previous branch.
    >>> t2 = make_treeil(2)
    >>> print_tree(t2)
    0
      0
        0
        1
        2
      1
        1
        2
        3
    """
    def treeil_tracker(k, parent):
        if n == k:
            return tree(parent)
        return _________
                   (a)
    return treeil_tracker(0, 0)
```

i. (0.0 pt) Fill in blank (a).

```
tree(parent, [treeil_tracker(k+1, i) for i in range(parent,
parent+k+2)])
```

No more questions.