CS61A MT2 Sp2018.

40.5/50

1. (10 points) Buggy Quidditch

For each of the expressions in the table below, 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.
- To display a function value, write **FUNCTION**.
- If an expression would take forever to evaluate, write **FOREVER**.

The interactive interpreter displays the contents of the `repr` string of the value of a successfully evaluated expression, unless it is `None`.

Assume that you have started `python3` and executed the code shown on the left first, then you evaluate each expression on the right in the order shown. Expressions evaluated by the interpreter have a cumulative effect.

```
class Ball:
    points = 0
    time = lambda: 'Draco'

    def score(self, who):
        print(who, self.points)

    def __str__(self):
        return 'Magic'

class Snitch(Ball):
    points = 100
    time = lambda: 'Harry'

    def __init__(self):
        self.points = self.points + 50

    def score(self, p):
        if not time():
            print(Ball().score(p))
        else:
            Ball.score(self, p)

def chase(r):
    r.time = Snitch.time
    r.points += 1
    quaffle.points += 10
    print(r().points)

quaffle = Ball()
quaffle.points = 10
chasing = quaffle.score
time = lambda: Ball.points
malfoy = lambda: Ball.time()
```

−2

Expression	Interactive Output
`Snitch().points`	150 ✓
`chasing(quaffle)`	Magic 10 ✓
`Snitch().score('Seeker')`	Seeker 150 ✗
`chase(Ball)`	1 ✓
`Snitch().score(malfoy())`	Harry 150 ✓

Class Ball: points [~~0~~ 1
time [→ fun λ

time [→ func λ [P=G]
malfoy [→ fn λ [P=G

Class Snitch(Ball): points [100
time [→ fun λ

chase [→ fun chase(r) [P=G]
quaffle [→ Ball instance · points : ~~10~~ 20
chasing [→ quaffle.score

Name: ____________________

-1

2. (6 points) NVRnment

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- **Use box-and-pointer diagrams for lists and tuples.**

```
s = 1
def to(s):
    ix = s[1:2]
    ward[1] = 6
    def re(st):
        if st is not ward:
            nonlocal s
            s = s.extend(ix)
            return re(ward)
        else:
            st.append(ix)
    return re
ward = [3, 4] + list([5])
to(ward)(s)
```

Global
s | 1
to | → func to(s) [parent=Global]
ward |

6
3 | ~~4~~ | ~~5~~ | 4

f1: to [parent=Global]
s |
ix | 4
re | func re(st) [Parent=f1]
Return Value |

f2: re [parent=f1]
st | 1
Return Value | None

f3: re [parent=f1]
st |
Return Value | None

f4: ________ [parent=________]
Return Value |

3. (12 points) Lists

Definition. A *grid* is a list of lists. Each list in a grid is called a *row*, and all rows must have the same length. `[[1, 2], [3, 4]]` is a grid of integers, but `[[1, 2], [3, 4, 5]]` is not a grid.

(a) (2 pt) Implement *column*, which takes a grid g and a non-negative integer c that is smaller than the length of a row in g. It returns a list containing the element at index c of each row in the grid.

```
def column(g, c):
    """Return the column of g at index c.

    >>> column([[3, 4, 5], [6, 7, 8], [9, 10, 11]], 1)
    [4, 7, 10]
    """
    return [ lst[c] for lst in g ]
```

(b) (4 pt) Implement print_grid, which takes a grid g. It prints one line for each row in g. Line k displays each element in row k, separated by the minimum number of spaces needed to align the left edges of the str strings for the elements of each column of the grid. At least one space should appear between any two elements. Extra spaces at the end of a line are ok. **You may not use the column function above. Assume that g has at least one row.**

```
def print_grid(g):
    """Print each row on a separate line with columns aligned.

    >>> print_grid([[1, 234, 50, 4, 5], [67, 8, 90, 0, 500], [3, 4, 5, -500, 7]])
    1   234 50 4    5
    67  8   90 0    500
    3   4   5  -500 7
    """
    cs = list(range(len(g[0])))

    widths = [ max ([len(str( row [ c ])) for row in g]) for c in cs]

    for row in g:

        line = ''

        for c in cs:

            s = str( row[c] )

            line = line + s + ' ' * ( widths[c] - len(s), )

        print(line)
```

-1

(c) (4 pt) Implement expand, which takes a grid g, a number of rows h, a number of columns w, and a fill value. It mutates the contents of g so that g has at least h rows and w columns. Any added values are fill.

```
def expand(g, h, w, fill):
    """Expand grid g so that it has at least h rows and w columns.

    >>> g = [[1, 2, 3], [40, 50, 60]]
    >>> print_grid(expand(g, 2, 5, 10))
    1  2  3  10 10
    40 50 60 10 10

    >>> print_grid(expand(g, 5, 6, 0))
    1  2  3  10 10 0
    40 50 60 10 10 0
    0  0  0  0  0  0
    0  0  0  0  0  0
    0  0  0  0  0  0

    >>> print_grid(expand(g, 0, 0, 5))    # expand never reduces the dimensions of g.
    1  2  3  10 10 0
    40 50 60 10 10 0
    0  0  0  0  0  0
    0  0  0  0  0  0
    0  0  0  0  0  0
    """

    for row in g:

        row.__extend__(__[fill] * (w - len(row))__)

    for k in __range( h - len(g))__:

        g.__append__(__[fill] * w__)

    return g
```

(d) (2 pt) Circle the Θ expression that describes how many new values must be added when a grid with n rows and n columns is expanded to $2 \times n$ rows and $2 \times n$ columns using the expand function. Assume that expand is implemented correctly.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$ None of these

4. **(12 points) Sequences**

(a) **(6 pt)** Implement `stretch`, which takes a `Link` instance `s` with no cycles. It mutates `s` so that, for each position k in the original `s`, the k^{th} element is repeated k times. **You do not need to use the name i.**

```
def stretch(s, repeat=0):
    """Replicate the kth element k times, for all k in s.

    >>> a = Link(3, Link(4, Link(5, Link(6))))
    >>> stretch(a)
    >>> print(a)
    <3, 4, 4, 5, 5, 5, 6, 6, 6, 6>
    """
    if s != Link.empty:

        for i in range(repeat):

            s = Link(s.first, s)                -1.5

            s = s.rest

        stretch(s.rest, repeat+1)
```

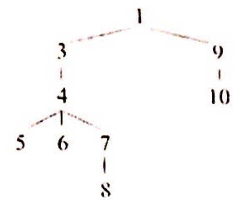
(b) **(6 pt)** Implement `combo`, which takes two non-negative integers `a` and `b`. It returns the smallest integer that contains all of the digits of `a` in order, as well as all of the digits of `b` in order.

```
def combo(a, b):
    """Return the smallest integer with all of the digits of a and b (in order).

    >>> combo(531, 432)       # 45312 contains both _531_ and 4_3_2.
    45312
    >>> combo(531, 4321)      # 45321 contains both _53_1 and 4_321.
    45321
    >>> combo(1234, 9123)     # 91234 contains both _1234 and 9123_.
    91234
    >>> combo(0, 321)         # The number 0 has no digits, so 0 is not in the result.
    321
    """
    if a == 0 or b == 0:

        return a + b

    elif a%10 == b%10:

        return combo(a//10, b//10) *10 + a%10

    return min(combo(a//10, b)*10 + a%10,
               combo(a, b//10)*10 + b%10)
```

5. (10 points) Trees

Definition. A *sibling* of a node in a tree is another node with the same parent.

(a) (4 pt) Implement `siblings`, which takes a `Tree` instance `t`. It returns a list of the labels of all nodes in `t` that have a sibling. These labels can appear in any order.




```
def siblings(t):
    """Return a list of the labels of all nodes that have siblings in t.

    >>> a = Tree(4, [Tree(5), Tree(6), Tree(7, [Tree(8)])])
    >>> siblings(Tree(1, [Tree(3, [a]), Tree(9, [Tree(10)])]))
    [3, 9, 5, 6, 7]
    """
    result = [b.label for b in t.branches if len(t.branches)≥2]

    for b in t.branches:

        result.extend(siblings(b))

    return result
```

(b) (6 pt) Implement the `Sib` class that inherits from `Tree`. In addition to `label` and `branches`, a `Sib` instance `t` has an attribute `siblings` that stores the number of siblings `t` has in `Sib` trees containing `t` as a node. **Assume that the branches of a `Sib` instance will never be mutated or re-assigned.**

```
class Sib(Tree):
    """A tree that knows how many siblings it has.

    >>> a = Sib(4, [Sib(5), Sib(6), Sib(7, [Sib(8)])])
    >>> a.label
    4
    >>> a.branches[1].label
    6
    >>> a.siblings
    0
    >>> a.branches[1].siblings
    2
    """
    def __init__(self, label, branches=[]):

        self.siblings = len(branches)

        for b in branches:

            assert isinstance(b, Sib)

        Tree.__init__(self, label, branches)
```

-4