INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except two hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A midterm 1 and midterm 2 study guides.
- Mark your answers on the exam itself. We will not grade answers written on scratch paper.

<table>
<thead>
<tr>
<th>Last name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td></td>
</tr>
<tr>
<td>Student ID number</td>
<td></td>
</tr>
<tr>
<td>CalCentral email (<a href="mailto:_@berkeley.edu">_@berkeley.edu</a>)</td>
<td></td>
</tr>
<tr>
<td>TA</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your right</td>
<td></td>
</tr>
</tbody>
</table>

All the work on this exam is my own. (please sign)

POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, `len`, and `abs`.
- You may not use example functions defined on your study guides unless clearly specified by the question.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
- You may use the `Tree`, `Link`, and `BTree` classes defined on Page 2 (left column) of the Midterm 2 Study Guide.
1. (10 points) **Buggy Quidditch** *(At least one of these is out of Scope: WWPD, Objects)*

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.

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

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

8 def __str__(self):
  return 'Magic'

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

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

25 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()
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snitch().points</td>
<td></td>
</tr>
<tr>
<td>chasing(quaffle)</td>
<td></td>
</tr>
<tr>
<td>Snitch().score('Seeker')</td>
<td></td>
</tr>
<tr>
<td>chase(Ball)</td>
<td></td>
</tr>
<tr>
<td>Snitch().score(malfoy())</td>
<td></td>
</tr>
</tbody>
</table>

In the code above, the class `Ball` is defined with attributes `points` and `time`, and methods `score` and `__str__`. The class `Snitch` inherits from `Ball` and overrides the `points` and `time` attributes, along with the `score` method. The `chase` function updates the `time` attribute of the `Ball` object and increments its `points` attribute along with the `Quaffle` object's `points`. The `malfoy` function returns the `points` attribute of the `Ball` object.
2. (6 points) **NVRnment** *(At least one of these is out of Scope: Environment Diagram, Lists, Nonlocal)*

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.

```
Global
 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)
```

<table>
<thead>
<tr>
<th>Frame</th>
<th>Function</th>
<th>Parent</th>
<th>Local</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td><code>to</code></td>
<td><code>Global</code></td>
<td><code>s</code></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><em>func</em></td>
<td></td>
<td><code>to</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>ward</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1: to</td>
<td></td>
<td><code>Global</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f3:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f4:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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) *(All are in Scope: Lists)* 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.

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

(b) (4 pt) *(All are in Scope: Lists)* 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.

```python
def print_grid(g):
    """Print each row on a separate line with columns aligned."
    cs = ___________________________________________________________________________________
    widths = [ __________ ([len(str( __________ [ ___________ ])) for row in g]) for c in cs]
    for row in g:
        line = ''
        for c in cs:
            s = str(______________________________)
            line = line + s + ' ' * (______________________________)
        print(line)
```
(c) (4 pt) \textit{(All are in Scope: Lists)} Implement `expand`, which takes a grid \(g\), a number of rows \(h\), a number of columns \(w\), and a \texttt{fill} value. It mutates the contents of \(g\) so that \(g\) has at least \(h\) rows and \(w\) columns. Any added values are \texttt{fill}.

```python
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.
    for k in :
        g.
    return g
```

(d) (2 pt) \textit{(At least one of these is out of Scope: Growth)} Circle the \(\Theta\) 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.

\[
\Theta(1) \quad \Theta(\log n) \quad \Theta(n) \quad \Theta(n^2) \quad \Theta(2^n) \quad \text{None of these}
\]
4. (12 points) Sequences

(a) (6 pt) (All are in Scope: Linked Lists) 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`.**

```python
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, 6>
    """
    if ______________________________________________________________________________________:
        for i in range(repeat):
            __________ = ________________________________________________________________
            __________ = ________________________________________________________________
            ______________________________________________________________________________________
```

(b) (6 pt) (All are in Scope: Tree Recursion) 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.

```python
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 _123_ and 9123_.
    91234
    >>> combo(0, 321)  # The number 0 has no digits, so 0 is not in the result.
    321
    """
    if _____________________________________________________________________________________:
        return a + b
    elif ____________________________________________________________________________________:
        return combo(___________________, __________________)_______________________________
    return ________(________________________________________________________________________,
                        __________________________________________________________________________)
5. (10 points) Trees

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

(a) (4 pt) (All are in Scope: Trees) 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.

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

    result = [_______________________________________________________________________________
    for b in t.branches:
        _______________________________________________________________________________________
    return result
```

(b) (6 pt) (All are in Scope: Objects, Trees) 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.

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

    def __init__(self, label, branches=[]):
        self.siblings = __________________________________________________________________________:
        ______________________________________________________________________________________:
        _________________________________________________________________________________
```

Tree.________________________________________________________________________________________