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 provided CS 61A 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 (@berkeley.edu)</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, abs, sum, next, iter, list, sorted, reversed, tuple, map, filter, zip, all, and any.
• You may not use example functions defined on your study guide unless a problem clearly states you can.
• 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 and Link classes defined on Page 2 (left column) of the Midterm 2 Study Guide.
1. (12 points) **What Would Python Display** *(At least one of these is out of Scope: WWPD, Nonlocal, Object-Oriented Programming, Python Lists)*

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. If evaluation would run forever, write “Forever”. To display a function value, write “Function”. The first two rows have been provided as examples.

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 first started python3 and executed the statements on the left, which cause no errors. Assume that expressions on the right are executed in the order shown. Expressions evaluated by the interpreter have a cumulative effect.

class C:
    x = 'e'
    def f(self, y):
        self.x = self.x + y
        return self
    def __str__ (self):
        return 'go'

class Big(C):
    x = 'u'
    def f(self, y):
        C.x = C.x + y
        return C.f(self, 'm')
    def __repr__ (self):
        return '<bears>'

m = C().f('i')
n = Big().f('o')

def g(y):
    def h(x):
        nonlocal h
        k = 2
        if len(x) == 1:
            h = lambda x: y
            k = 1
        return [-x[0]] + h(x[k:])
    return h(y)

py = {3: {5: [7]}}
thon = {5: 6, 7: py}
thon[7][3] = thon

q = range(3, 5)
r = iter(q)
code = [next(r), next(iter(q)), list(r)]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow(10, 2)</td>
<td>100</td>
</tr>
<tr>
<td>print(4, 5) + 1</td>
<td>4 5</td>
</tr>
<tr>
<td>[m.x, n.x]</td>
<td></td>
</tr>
<tr>
<td>[C.f(n, 'a').x, C().x]</td>
<td></td>
</tr>
<tr>
<td>print(m, n)</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>g([3, 4, 5])</td>
<td></td>
</tr>
<tr>
<td>py[3][5]</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
</tr>
</tbody>
</table>
2. (8 points) Environmental Studies *(At least one of these is out of Scope: Environment Diagrams, Python Lists, Nonlocal)*

Fill in the environment diagram that results from executing the code on the right 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:

- Use box-and-pointer notation for lists.
- Add all missing names and parent annotations.
- Add all missing values created or referenced.
- Show the return value for each local frame.

```python
def power(cut):
    def wind(y):
        nonlocal cut
        cut, g = cut - 1, cut
        y = [e]
        return y is e
    if cut and not wind('y'):
        return power(cut)
    else:
        return [g, 'e']
g = 'p'
e = ['p']
e.append(power(1))
```

```
Global frame power

def power(cut):
    def wind(y):
        nonlocal cut
        cut, g = cut - 1, cut
        y = [e]
        return y is e
    if cut and not wind('y'):
        return power(cut)
    else:
        return [g, 'e']
ge = 'p'
e = ['p']
e.append(power(1))
```
3. (6 points) Max Tree *(All are in Scope: Trees, Tree Recursion)*

(a) (4 pt) Implement `max_tree`, which takes a `Tree` instance `t` and a function `key` that takes one argument and returns a number. The `max_tree` function returns the label `n` of `t` for which `key(n)` is largest. If there is more than one label for which `key` returns the same largest value, `max_tree` can return any of those labels. The `Tree` class is defined on the Midterm 2 Study Guide. You may not call `min` or `max`.

```python
def max_tree(t, key):
    """Return the label n of t for which key(n) returns the largest value."

    >>> t = Tree(6, [Tree(3, [Tree(5)]), Tree(2), Tree(4, [Tree(7)])])
    >>> max_tree(t, key=lambda x: x)
    7
    >>> max_tree(t, key=lambda x: -x)
    2
    >>> max_tree(t, key=lambda x: -abs(x - 4))
    4
    """
    if t.is_leaf():
        return ___________________________________________________________________________
    x = __________________________________________________________________________________
    for b in t.branches:
        m = ______________________________________________________________________________
        if _______________________________________________________________________________:
            x = m
    return x
```

(b) (2 pt) Now implement `max_tree` in one line using a call to the built-in `max` function. You may not include any additional calls to `max` beyond the one included in the template below.

```python
def max_tree(t, key):
    """Return the label n of t for which key(n) returns the largest value."

    return max(____________________________________________________________________________, key=key)
```
4. (4 points) Seek Once *(All are in Scope: Python Lists)*

Implement `stable`, which takes a list of numbers `s`, a positive integer `k`, and a non-negative number `n`. It returns whether all pairs of values in `s` with indices that differ by at most `k` have an absolute difference in value of at most `n`.

You may not use `lambda`, `if`, `and`, or `or` in your solution.

**Note:** The “functions that aggregate iterable arguments” on the Midterm 2 Study Guide have the following behavior when called on an empty list:

- `sum([])` evaluates to 0.
- `max([])` causes a `ValueError`.
- `min([])` causes a `ValueError`.
- `all([])` evaluates to `True`.
- `any([])` evaluates to `False`.

```python
def stable(s, k, n):
    """Return whether all pairs of elements of S within distance K differ by at most N."""

    for i in range(len(s)):
        near = range(i-k, i)
        if ________________([ ________________ > n for j in near]):
            return False

    return True
```

```python
>>> stable([1, 2, 3, 5, 6], 1, 2)  # All adjacent values differ by at most 2.
True
>>> stable([1, 2, 3, 5, 6], 2, 2)  # abs(5-2) is a difference of 3.
False
>>> stable([1, 5, 1, 5, 1], 2, 2)  # abs(5-1) is a difference of 4.
False
"""
```
5. (7 points) Do You Yield?

(a) (6 pt) (All are in Scope: Tree Recursion, Iterators, Generators) Implement partitions, which is a generator function that takes positive integers \( n \) and \( m \). It yields strings describing all partitions of \( n \) using parts up to size \( m \). Each partition is a sum of non-increasing positive integers less than or equal to \( m \) that totals \( n \). The partitions function yields a string for each partition exactly once.

You may not use lambda, if, and, or, lists, tuples, or dictionaries in your solution (other than what already appears in the template).

```python
def partitions(n, m):
    
    >>> list(partitions(1, 1))
    ['1']
    >>> list(partitions(2, 2))
    ['2', '1 + 1']
    >>> list(partitions(4, 2))
    ['2 + 2', '2 + 1 + 1', '1 + 1 + 1 + 1']
    >>> for p in partitions(6, 4):
    ...     print(p)
    ...     print(p)
    4 + 2
    4 + 1 + 1
    3 + 3
    3 + 2 + 1
    3 + 1 + 1 + 1
    2 + 2 + 2
    2 + 2 + 1 + 1
    2 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 1
    
    if _____________________________:
        yield _____________________________

    if n > 0 and m > 0:

        for p in partitions(______________, ________________):

            yield _____________________________

        yield from partitions(______________, ________________)
```

(b) (1 pt) (At least one of these is out of Scope: Asymptotic Notation) Circle the order of growth of the value of \( \text{len(list(partitions(n, 2)))} \) as a function of \( n \). For example, the value of \( \text{len(list(partitions(3, 2)))} \) is 2, because there are 2 partitions of 3 using parts up to size 2: 2 + 1 and 1 + 1 + 1. Assume partitions is implemented correctly.

Constant  Logarithmic  Linear  Quadratic  Exponential  None of these
6. (4 points) Best of Both *(All are in Scope: Python Lists, Tree Recursion)*

Definition. A switch list $r$ for two source lists $s$ and $t$, both of length $n$, is a list where each element $r[i]$ for $0 \leq i < n$ is either $s[i]$ or $t[i]$. The switch count for $r$ is the number of indices $i$ for which $r[i]$ and $r[i-1]$ come from different lists. As a special case, index 0 contributes 0 to the switch count if $r[0]$ comes from $s$ and 1 if it comes from $t$.

Implement `switch`, which takes two lists of numbers $s$ and $t$ that have the same length, and a non-negative integer $k$. It returns the switch list for $s$ and $t$ that has the largest sum and has a switch count of at most $k$.

Constraints:
- The blanks on line 4 must be filled with one of the following combinations:
  - $s$, $t$
  - $t$, $s$
  - $s[1:]$, $t$
  - $t$, $s[1:]$
  - $s[1:]$, $t[1:]$
  - $t[1:]$, $s[1:]$
- The blank on line 5 must use one of the following templates:
  - _____ + switch(_______, ________, k - 1)
  - _____ + switch(_______, ________, k)
  - switch(_______, ________, k - 1)
  - switch(_______, ________, k)

```python
def switch(s, t, k):
    """Return the list with the largest sum built by switching between S and T at most K times."

    >>> switch([1, 2, 7], [3, 4, 5], 0)
    [1, 2, 7]
    >>> switch([1, 2, 7], [3, 4, 5], 1)
    [3, 4, 5]
    >>> switch([1, 2, 7], [3, 4, 5], 2)
    [3, 4, 7]
    >>> switch([1, 2, 7], [3, 4, 5], 3)
    [3, 4, 7]
    """
    if k == 0 or len(s) == 0:
        return s
    else:
        a = switch(_______, ________, k-1)
        b = ________________________________
        return max(a, b, key=sum)
```

7. **(5 points) Version 2.0** *(All are in Scope: Object-Oriented Programming, Python Lists)*

Implement the `Version`, `Insert`, and `Delete` classes as follows:

- A `Version` represents a string that has been edited. A `Version` instance has a `previous` value (which may be a `str` or `Version` instance) and an `edit` (which may be an instance of `Insert` or `Delete`).
- Calling `str` on a `Version` instance should return the string that results from applying its `edit` to the string represented by `previous`. Assume that string is long enough to perform the `edit`.
- An `edit` object represents a change to a string parameterized by some value `c` applied at an index `i`.
- The `apply` method for `Insert` inserts the string `c` into string `t` starting at index `i`.
- The `apply` method for `Delete` removes `c` (a positive `int`) characters from string `t` starting at index `i`.

class Version:
    """A version of a string after an edit.
    >>> s = Version('No power?', Delete(3, 6))
    >>> print(Version(s, Insert(3, 'class!')))
    No class!
    >>> t = Version('Beary', Insert(4, 'kele'))
    >>> print(t)
    Bearkeley
    >>> print(Version(t, Delete(2, 1)))
    Berkeley
    >>> print(Version(t, Delete(4, 5)))
    Bear"
    def __init__(self, previous, edit):
        self.previous, self.edit = previous, edit
    def __str__(self):
        return ___________________________________________________________________________________

class Edit:
    def __init__(self, i, c):
        self.i, self.c = i, c

class Insert(Edit):
    def apply(self, t):
        "Return a new string by inserting string c into t starting at position i."
        return ____________________________________________________________________________________

class Delete(Edit):
    def apply(self, t):
        "Return a new string by deleting c characters from t starting at position i."
        return ____________________________________________________________________________________
8. (4 points) Their Mascot is a Tree?!? (All are in Scope: Trees, Linked Lists)

Definitions. The depth of the root node of a tree is 0, and the depth of a child is 1 more than the depth of its parent. A linked list is either a Link instance or Link.empty.

Implement layer, which takes a Tree instance t and a non-negative integer d. It returns a linked list containing the labels of all nodes in t that have depth d. Labels in the result should appear in branch order. The Tree and Link classes appear on the Midterm 2 Study Guide.

You may not use lambda, if, and, or in your solution (other than what already appears in the template).

Note: The built-in reversed function returns a reverse iterator over the values of a sequence. For example, list(reversed([1, 2, 3])) evaluates to [3, 2, 1].

```python
def layer(t, d):
    # Return a linked list containing all labels of Tree T at depth D.
    >>> a_tree = Tree(1, [Tree('b', [Tree('mas')]),
                         ... Tree('a', [Tree('co')]),
                         ... Tree('d', [Tree('t', [Tree('!')])])])
    >>> print(layer(a_tree, 0))
    <1>
    >>> print(layer(a_tree, 1))
    <b a d>
    >>> print(layer(a_tree, 2))
    <mas co t>
    >>> print(layer(a_tree, 3))
    <!>
    ""
    return helper(t, d, _____________________________________________________________)

def helper(t, d, s):
    if d == 0:
        return _____________________________________________________________
    else:
        for b in reversed(t.branches):
            s = _____________________________________________________________
        return s
```