Question 1. [5 MARKS]

Read over the definition of this Python function:

```
def i(n):
    """Docstring (almost) omitted."""
        return 1 + sum([i(j) for j in n]) if isinstance(n, list) else 0
```

Work out what each function call produces, and write it in the space provided.

1. `i(5)`
   0

   0
   1 + sum([i(j) for j in []])
   1 +sum([0])
   1

   1 + sum([i(j) for j in [1,2,3]])
   1 + sum([0,0,0])
   1

2. `i([])`
   1

3. `i([1, 2, 3])`
   1

   1 + sum([i(j) for j in [1, [2, 3], 4, [5, 6]]])
   1 + sum([0, 1 + sum([i(j) for j in [2,3]]),0, 1 + sum([i(j) for j in [5,6]])])
   1 + sum([0,1+0,0,1+0])
   1+ sum([0,1,0,1])
   3

4. `i([1, [2, 3], 4, [5, 6]])`
   3

5. `i([1, [2, 3, [3.5]], 4, [5, 6, [7, 8]]])`
   5

Question 2. [5 MARKS]

Read over the declarations of the three **Exception** classes, the definition of **raiser**, and the supplied code for **notice** below. Then complete the code for **notice**, using only **except** blocks, and perhaps an **else** block.

```
class EX(Exception):
    pass

class EXX(EX):
    pass

class EXXX(EXX):
    pass

def raiser(n: int) -> None:
    """Raise exceptions based on divisibility of n"""
    if n % 12 == 0:
        raise EXXX
    elif n % 6 == 0:
        raise EXX
    elif n % 3 == 0:
```

```
        raise EX
    else:
        b = 1 / n

def notice(n: int) -> str:
    """Return message appropriate to raiser(n).

    >>> notice(17)
    'fine'
    >>> notice("compute")
    'whatever!'
    >>> notice(12)
    'oops! oops! oops!'
    >>> notice(6)
    'oops! oops!'
    >>> notice(3)
    'oops!'
    """
    try:
        raiser(n)
    # Write some "except" blocks and perhaps an "else" block
    # below that make notice(...)
    # have the behaviour shown in the docstring above

    except EXXX:
        return 'oops! oops! oops!'
    except EXX:
        return 'oops! oops!'
    except EX:
        return 'oops!'
    except Exception:
        return 'whatever!'
    else:
        return 'fine'
```

Question 3. [5 MARKS]

Read over the declaration of the class **Tree** and the docstring of the function **initial_a_count**. Then complete the implementation of **initial_a_count**

```
class Tree:
    """Bare-bones Tree ADT"""

    def __init__(self: 'Tree',
                 value: object =None, children: list =None):
```

```
        """Create a node with value and any number of children"""

        self.value = value
        if not children:
            self.children = []
        else:
            self.children = children[:] # quick-n-dirty copy of list


def initial_a_count(t: Tree) -> int:
    """Return number of values in t that begin with "a"

    precondition - t is a non-empty tree with non-empty string values

    >>> tn2 = Tree("one", [Tree("two"), Tree("three"),\
Tree("apple"), Tree("five")])
    >>> tn3 = Tree("answer", [Tree("six"), Tree("seven")])
    >>> tn1 = Tree("eight", [tn2, tn3])
    >>> initial_a_count(tn1)
    2
    >>> initial_a_count(tn2)
    1
    """

    return (sum([initial_a_count(c) for c in t.children]) +
            (1 if t.value[0] == 'a' else 0))
```

Question 4. [5 MARKS]

Complete the implementation of **push** in the class **AlphaStack**, a subclass of **Stack**. Notice that you may use **push**, **pop**, and **is_empty**, the public operations of **Stack**, but you may not assume anything about **Stack**'s underlying implementation. You may find it useful to know that if **s1** and **s2** are strings, then **s1** $<$ **s2** is True if and only if **s1** is smaller than **s2** in alphabetical order.

```
from csc148stack import Stack
"""
Stack operations:
    pop(): remove and return top item
    push(item): store item on top of stack
    is_empty(): return whether stack is empty.
"""

class AlphaStack(Stack):
    """Stack of strings in descending alphabetical order"""
```

```
def push(self: 'AlphaStack', s: str) -> None:
    """Place s on top of stack self provided it is smaller
    in alphabetic order than the string currently on top of
    stack self (if there is one).  Otherwise raise an Exception
    and leave stack self as it was.

    precondition - possibly empty self contains only strings

    >>> s = AlphaStack()
    >>> s.push("behemoth")
    >>> s.push("asterisk")
    >>> # now s.push("caliph") should raise Exception
    """

    if not self.is_empty():
        last = self.pop()
        Stack.push(self, last)
        if  last <= s:
            raise Exception(
                '{} is not alphabetically lower than {}'.format(s, last))
    Stack.push(self, s)
```