Question 1. [5 MARKS]

Read over the definition of this Python function:

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

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

1. `c(5)`

   `0`

2. `c([])`

   `0`

3. `c([1, 3, 5])`

   `3`

4. `c([0, [1, 3, 5], 7])`

   `3`

5. `c([0, [1, 3, 5, [7, [9]]], 11])`

   `4`

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 E1(Exception):
    pass

class E2(E1):
    pass

class E3(E2):
    pass

def raiser(n: int) -> None:
    """Raise exceptions based magnitude of n"""
    if n < 2:
        raise E3
    elif n < 4:
        raise E2
    elif n < 6:
```

```
        raise E1
    else:
        b = 1 / n

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

    >>> notice(15)
    'ok'
    >>> notice("CSC148")
    'purple alert!'
    >>> notice (1)
    'red alert!'
    >>> notice(3)
    'orange alert!'
    >>> notice(5)
    'yellow alert!'
    """
    try:
        raiser(n)
    # Write some "except" blocks and perhaps an "else" block
    # below that make notice(...)
    # have the behaviour shown the the docstring above

    except E3:
        return 'red alert!'
    except E2:
        return 'orange alert!'
    except E1:
        return 'yellow alert!'
    except Exception:
        return 'purple alert!'
    else:
        return 'ok'
```

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**. You may find the builtin Python function **sum(L)** useful, which returns the sum of the numbers in list **L**, or 0 if **L** is empty.

```
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 **PrefixStack**, 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.startswith(s2)** returns True if **s2** is a prefix of **s1**, and False otherwise.

```
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 PrefixStack(Stack):
```

```
    """Stack of strings where each is a prefix of its predecessor"""

    def push(self: 'PrefixStack', s: str) -> None:
        """Place s on top of stack self, provided s is a prefix of
        its predecessor.  Otherwise raise an Exception and leave
        stack self as it was

        precondition - possibly empty self contains only strings

        >>> s = PrefixStack()
        >>> s.push("asterisk")
        >>> s.push("aster")
        >>> # now s.push("asteri") should raise Exception
        """

        if not self.is_empty():
            last = self.pop()
            Stack.push(self, last)
            if  not last.startswith(s):
                raise Exception('{} not a prefix of {}'.format(s, last))
        Stack.push(self, s)
```