Started on	Sunday, 31 January 2021, 9:06 AM
State	Finished
Completed on	Sunday, 31 January 2021, 10:15 AM
Time taken	1 hour 9 mins
Grade	Not yet graded

Question **1**

Complete

Marked out of 3.00

*A sparse data structure is one where we presume most of the elements have a common value (e.g. 0). Write the **SparseList** class, which implements a sparse list data structure, so that the following code works as specified by the comments. Each comment refers to the line(s) below it. Elements not explicitly set will have the default value 0. The list's length is given by the element set using the largest index (hint: use the **__setitem__** and **__getitem__** methods). Do not represent 0 values in memory! Specifications or tests are not required **[3p]**.*

```
# Initialise the sparse list
data = SparseList()
# Add elements to the sparse list
data[1] = "a"
data[4] = "b"
data[9] = "c"
# Element at index 14 is 'd'
data[14] = "d"
# append adds the element after the last
# initialised index
data.append("z")
# Prints:
# 0 a 0 0 b 0 0 0 0 c 0 0 0 0 d z
for i in range(0, len(data)):
    print(data[i])
```

```
###THIS IS THE BAD STUFF MEMORISING ZEROS###
```

```
class SparseList:
    def __init__(self):
        self.elems = []

    def __setitem__(self, key, item):
        while len(self.elems) <= key:
            self.elems.append(0)
        self.elems[key] = item

    def __getitem__(self, key):
        return self.elems[key]

    def append(self, value):
        self.elems.append(value)

    def __len__(self):
        return len(self.elems)
```

###THIS IS THE GOOD STUFF NOT MEMORISING ZEROS###

```
class SparseList:
    def __init__(self):
        self.elems = {}
        self.last_index = 0

    def __setitem__(self, key, item):
        if key > self.last_index:
            self.last_index = key
        self.elems[key] = item

    def __getitem__(self, key):
        if key in self.elems.keys():
            return self.elems[key]
        else:
            return 0

    def append(self, value):
        self.last_index += 1
        self.elems[self.last_index] = value

    def __len__(self):
        return self.last_index


# Initialise the sparse list
data = SparseList()
# Add elements to the sparse list
data[1] = "a"
data[4] = "b"
data[9] = "c"
# Element at index 14 is 'd'
data[14] = "d"
# append adds the element after the last
# initialised index
data.append("z")
# Prints:
# 0 a 0 0 b 0 0 0 0 c 0 0 0 0 d z
for i in range(0, len(data)):
    print(data[i])
```

Question **2**

Complete

Marked out of 2.00

Analyse the time and extra-space complexity of the following function **[2p]**.

```
def f(a,b):
    if (a!=1):
        f(a-1,b-1)
        print (a,b)
        f(a-1,b-1)
    else:
        print (a,b)
```

```
"""
The recursive calls of the given function depends on the variable 'a';
TIME COMPLEXITY
T(a) = 2T(a-1)+1, a>1
       0, a=1
       +infinity, a<1 (calling f(a-1,b-1) when a<1 will never result in a being 1 for the calls to stop)

T(a) = 2T(a-1)+1        | *2^0
T(a-1) = 2T(a-2)+1      | *2^1
T(a-2) = 2T(a-3)+1      | *2^2
.
.
T(2) = 2T(1)+1          | *2^(a-2)
T(1) = 0                | *2^(a-1)
---------------------------------- +
we are left with T(a) = 2^0 + 2^1 + 2^2 + ... + 2^(a-2) = 1*(2^(a-1) -1)/(2-1) = 2^a (approx) => O(2^a)

SPACE COMPLEXITY
For each recursive call, a new stackframe is created, so the stack memory is affected
The number objects are created in the heap memory, and the variables a and b from the stack point to them
Since there are 2 identical calls ( f(a - 1, b - 1) ), the number objects are created in the heap only once
So space complexity for the heap memory would linear, T(a)=2a (for numbers corresponding to vars a and b) => O(a)
Stack memory is O(2^a) because each recursive call has its own a and b in the stackframe
"""
```

Question **3**

Complete

Marked out of 4.00

Write the specification, Python code and test cases for a ***recursive function*** that uses the ***divide and conquer*** method to calculate the sum of the even numbers found on even positions in a list. The function should return ***None*** if no suitable numbers are found. For the input [**2**, 2, **4**, 5, **6**, 4, 13, 4, **10**], the result is 22. You may divide the implementation into several functions according to best practices. Specify and test all functions **[4p]**.

```
"""
Write the specification, Python code and test cases for a recursive function that uses the divide and conquer method
to calculate the sum of the even numbers found on even positions in a list. The function should return None if no
suitable numbers are found. For the input [2, 2, 4, 5, 6, 4, 13, 4, 10], the result is 22. You may divide the
implementation into several functions according to best practices. Specify and test all functions [4p].
"""

"""
     0, 1, 2, 3, 4, 5, 6,  7, 8
v = [2, 2, 4, 5, 6, 4, 13, 4, 10]

    [2, 2, 4, 5] 6 [4, 13, 4, 10]

    [2, 2][4, 5]   [4, 13][4, 10]
"""


def compute_even_sum(x):
    """ not working yet :( but the version below works
    Computes the sum of even numbers on even positions by using divide and conquer
    the list is divided in halves (while keeping the parity of the indexes)
    base_cases: length is 3, when we check the first and last elems
                length is 2, when we check the first elem
    combine: the result is added to the return value of each recursive call
        (return x[i] + compute_even_sum(x[:i + 1]) + compute_even_sum(x[i + 1:]))
    :param x: list of numbers
    :return: integer number representing sum of even numbers on even positions or None
    """
    if len(x) == 2:
        if x[0] % 2 == 0:
            return x[0]
    if len(x) == 3:
        ts = 0
        if x[0] % 2 == 0:
            ts += x[0]
        if x[2] % 2 == 0:
            ts += x[2]
        return ts
    if len(x) >= 3 and len(x) % 2 == 1:  # odd
        i = len(x) // 2
        if i % 2 == 0 and x[i] % 2 == 0:
            return x[i] + compute_even_sum(x[:i + 1]) + compute_even_sum(x[i + 1:])
        else:
            return compute_even_sum(x[:i + 1]) + compute_even_sum(x[i + 1:])
    elif len(x) >= 2 and len(x) % 2 == 0:
        i = len(x) // 2
        return compute_even_sum(x[:i + 1]) + compute_even_sum(x[i + 1:])
    else:
        return None


def compute_sum_working(x):
    """
    Computes sum of even numbers on even positions
    kind of chip and conquer, because it only divides the list into n-2 elems

    :param x: list of numbers
    :return: integer number representing sum of even numbers on even positions
    """
    if len(x) == 0:
        return 0
    if len(x) == 1:
        return x[0] if x[0] % 2 == 0 else 0
    return x[0] + compute_sum_working(x[2:]) if x[0] % 2 == 0 else compute_sum_working(x[2:])
```

```
"""
TESTSSSS
"""
from written_exam import compute_sum_working


class Tests(unittest.TestCase):
    def setUp(self):
        self.l1 = [2, 2, 4, 5, 6, 4, 13, 4, 10]
        self.l2 = [1, 1, 1, 1]
        self.l3 = [5, 4]
        self.l4 = [2]
        self.l5 = [2, 3, -2]
        self.not_list = 5

    def test_sums(self):
        self.assertEqual(compute_sum_working(self.l1), 22)
        self.assertEqual(compute_sum_working(self.l2), 0)  # should be None
        self.assertEqual(compute_sum_working(self.l3), 0)  # should be None
        self.assertEqual(compute_sum_working(self.l4), 2)
        self.assertEqual(compute_sum_working(self.l5), 0)

        self.assertRaises(TypeError, compute_sum_working, self.not_list)
```

◄ Practice Quiz

Jump to...