Home / My courses / FP / Exam / Written Exam

Started on	Sunday, 31 January 2021, 9:05 AM
State	Finished
Completed on	Sunday, 31 January 2021, 10:13 AM
Time taken	1 hour 8 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 commo implements a sparse matrix data structure, so that the following code works as specifie below it. Matrix elements not explicitly set have the default value 0. The matrix size is p memory! Specifications or tests are not required **[3p]**.*

```
# Initialise a 3x3 sparse matrix
m1 = SparseMatrix(3,3)
# Value at [1,1] is 2
m1.set(1,1,2)
# Value at [2,2] is 4
m1.set(2,2,4)
# Prints
# 0 0 0
# 0 2 0
# 0 0 4
print(m1)
# Prints '<class 'ValueError'>'
try:
    m1.set(3,3,99)
except Exception as e:
    print(type(e))
# Update value at [1,1] with 2 + 1
m1.set(1,1,m1.get(1,1)+1)
# Prints
# 0 0 0
# 0 3 0
# 0 0 4
```

```
class SparseMatrix:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.cells = {}     # dictionary: sparse

    def _encode(self, line, column):
        return line * self.height + column

    def _is_inside(self, line, column):
        return 0 <= line < self.height and 0 <= column < self.width

    def set(self, line, column, value):
        if not self._is_inside(line, column):
            raise ValueError("Out of bounds!")
        self.cells[self._encode(line, column)] = value
        if value == 0:
            del self.cells[self._encode(line, column)]

    def get(self, line, column):
        if not self._is_inside(line, column):
            raise ValueError("Out of bounds!")
        return self.cells.get(self._encode(line, column), 0)

    def __str__(self):
        output = ""
        for line in range (self.height):
            for column in range(self.width):
                output += str(self.get(line, column)) + " "
            if line != self.height - 1:
                output += "\n"
        return output
```

Question **2**

Complete

Marked out of 2.00

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

```
def f(n):
    s = 0
    for i in range(1, 3 ** n + 1):
        j = 1
        while j < n:
            s = s + j
            j *= 3
    return s
```

Time complexity:

The outer loop gets executed 3^n times (for i in range(1, 3 ** n + 1): interval [1, 3 **
inside the loop.

The inner loop's execution (while statement) doesn't depend on the outer loop varia
during execution. The value of j gets multiplied by 3 until it becomes greater than n,
the while statement fails (k=log3n+1 is the first number for which 3^k > n).

Total time complexity is O(3^n * logn).

Space complexity:

Variables i and j are used for loop counters, and s as a sum counter. The space comp
of memory through recursive or other data structures (j is declared at each step insid
every iteration, so space complexity remains constant).

Question **3**

Complete

Marked out of 4.00

Write the specification, Python code and test cases for a **recursive function** which u
calculate the sum of the numbers found on prime positions in a list of natural numbe
15, **16**, 17, **18**, 19] will return 7 + 8 + 10 + 12 + 16 + 18 = **71**. Divide the implementa
Specify and test all functions **[4p]**.

```
import unittest


def is_prime(number):
    """
    Checks if the given number is a prime number.

    :param number:  natural number (number > 0)
    :return:        True if given number is prime, False otherwise
    """

    if number < 2:
        return False
    divisor = 2
    while divisor*divisor <= number:
        if number % divisor == 0:
            return False
        divisor += 1
    return True


def sum_of_prime_position(number_list, index=0):
    """
    Computes the sum of all numbers found on prime position in the given numb
    given position index (inclusively).

    :param number_list: integer list
    :param index:       integer >= 0
    :return:            sum of numbers on prime positions to the right of par
    """

    # base case
    if index >= len(number_list):
        return 0

    # chip and conquer
    recursive_case = sum_of_prime_position(number_list, index+1)
    current_case = number_list[index] if is_prime(index) else 0
    combine = recursive_case + current_case
    return combine
```

```
class TestModule(unittest.TestCase):
    def test_prime(self):
        self.assertEqual(is_prime(0), False)
        self.assertEqual(is_prime(1), False)
        self.assertEqual(is_prime(2), True)
        self.assertEqual(is_prime(3), True)
        self.assertEqual(is_prime(4), False)
        self.assertEqual(is_prime(5), True)
        self.assertEqual(is_prime(6), False)
        self.assertEqual(is_prime(7), True)
        self.assertEqual(is_prime(8), False)
        self.assertEqual(is_prime(9), False)
        self.assertEqual(is_prime(10), False)
        self.assertEqual(is_prime(11), True)
        self.assertEqual(is_prime(37), True)
        self.assertEqual(is_prime(102), False)

    def test_sum(self):
        self.assertEqual(sum_of_prime_position([5, 6, 7, 8, 9, 10, 11, 12, 13
```

◄ Practice Quiz

Jump to...