CS1010X — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

27 March 2021 **Time allowed:** 2 hours

Student No:

S	O	L	U	T	I	O	N	S

Instructions (please read carefully):

1. Write down your student number on the **question paper and all pages of the provided answer sheets**. DO NOT WRITE YOUR NAME ON THE QUESTION PAPER AND ANSWER SHEETS !
2. This is **an open-book test**.
3. Please switch off your mobile phone, and no laptop and no calculator. If found using any of these equipment any moment from now till your scripts have been collected, you will be given a zero mark.
4. This paper comprises **FOUR (4) questions** and **ELEVEN (11) pages**. The time allowed for solving this test is **2 hours**.
5. The maximum score of this test is **32 marks**. The weight of each question is given in square brackets beside the question number.
6. All questions must be answered in the space provided in the answer sheets (not this question paper). At the end of the test, please submit both your question paper and your answer sheets. No extra sheets will be accepted as answers.
7. You are allowed to use pencils. Please be sure your handwriting is legible, and you have proper indentation as needed for your Python codes.

GOOD LUCK!

Question	Marks
Q1	
Q2	
Q3	
Q4	
Total	

Question 1: Warm up - fill in the blank [1 marks]

Our (Python) computer program consists of 3 main constructs: sequential statements, conditional and repetition / looping statements.

Question 2: Python Expressions [9 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, one or more Python expressions are entered into the interpreter (Python shell). Determine all the response printed by the interpreter for the expressions entered and **write the exact output in the answer box**.

It is assumed that there is no syntax errors in our input to the Python interpreter. If the interpreter produces an execution error message at some stage, state where and the nature of the error (you don't need to show the exact error message but still need to write out any output from the Python expressions before the error was encountered). In another extreme, if the interpreter enters an infinite loop, explain how it happens. **Partial marks may be awarded for working if your answer is wrong. You are, however, responsible to make clear the part of your answer verse the part of your working.**

A. [2 marks]

```
def f(x):
    if x == 0:
        print("ready")
        return y
    elif x == 1 or y == 0:
        print("relax...")
    elif y == 1:
        print("go")
    x = x - 1
    f(x)
    return False

x = 2
y = 1
y = f(y)
print(not f(y+x))
```

Answer:

```
relax...
ready
relax...
relax...
ready
True
```

Note: A taste of tracing with many elements. Need a table on x, and y in the main routine, then a trace of x in the function and a stack to keep track of function calls. Also, note that `False` is equivalent to 0, boolean when used in addition is converted to 0 or 1 (as shown during our recitation). We have tried our best to give partial credits but there is a limit on what we can allow.

B. Below is a single statement entered into Python shell. [1 marks]

```
120,000 > 1,000
```

Answer:
`(120, False, 0)`

Note: This is another give away if you have attended our recitation. Some of you treated this as if 120 > 1 and then 000 > 000 (so, they wrote `True, False` as their answers) because they recalled Python has the assignment of `x,y = y,x` (for switching values) – unfortunately, "=" is an assignment, while ">" is a comparator (similar to "==").

C. [2 marks]

```
def f(x):
    if x==0:
        return 0
    result = 0
    for x in range(20,0,-2):
        x = 10
        result += x
    print (result)

x=3
y=1
x = f(x)
print (not x)
```

Answer:
```
100
True
```

Note: The change to the value of the controlling variable *x* in an iteration does not affect the assignment of its value in the following iteration. Also, a function that returns nothing is to return `None`. And, printing `not None` is to print a `True`.

D. Below are 8 statements entered into Python shell. [2 marks]

```
word = "0101010101010101010101" # len(word) = 22
mask = "-------*------*-------" # 7 "-" then "*" then 6 "-" then "*" then 7 "-"
masking = ( (11,-11), (6, -6), (3, -3), (2, -2), (1, -1) )

for i in range (0, 5):
    word = word[0::2]+word[1::2]
    maskedWord = word[0:masking[i][0]] + mask[masking[i][0]: masking[i][1]] + \
                 word[masking[i][1]:]
    print(maskedWord)
```

Answer: (please align your output pattern nicely in a grid to avoid penalty)

```
0000000000011111111111
000000-*------*-111111
000----*------*----111
00-----*------*-----11
0------*------*------1
```

Note: This is a test on the use of strings, and tuples.

E. [2 marks]

```
def comp(f, g):
    return lambda g: f(f(g))

def add2(x):
    return x+2

def add3(x):
    return lambda x: x+3

print ( add3(add2)(8) )
print ( add3((add2)(8))(88) )
print ( comp(add3, add2)(8)(88) )
print ( comp( add3(add2), add3(add2) )(8))
```

Answer:

```
11
91
91
14
```

Note: Only the last print statement needs good thinking. The second parameter to `comp` is not useful, so it can immediately be removed to think `comp` has just 1 parameter.

Question 3: Iteration & Recursion, with Time & Space [10 marks]

Suppose you are given the following functions:

```
def A(n):
    if n == 1:
        return 1
    else:
        return n + B(n+1)

def B(n):
    if n == 1:
        return 1
    else:
        return n + A(n//2)
```

A. What is the output value of `A(50)`? [1 marks]

209
Note: This part is here to help you understand the execution of `A` and `B` so as to prepare you for the next part. Partial credit is given if we see some correct working (while your additions turn out to be wrong without a calculator).

B. Provide the time and space complexity of the function `A` in terms of input `n`. Justify your answer with consideration of the details of the given programs. [3 marks]

Time: $O(\log n)$
Space: $O(\log n)$
Justification (for time and for space):

We assume n is a non-negative number. (1) You should argue about the increment by 1 to n in

`B` does not alter the fact that each time `A` is in turn called again, the value of `n` is at most half (of what it was before) plus 1. This is the same for `B` when the function execution returns to `B` the value of `n` is at most half plus 1 too.

(2) When n is positive, they are taken care of by the above argument. Then, what happen if $n = 0$ (and the programs may not terminate as a result)? For `A`(0), it calls `B`(1), and `B`(1) returns immediately with 1 and thus `A`(0) has no issue. For `B`(0), it calls `A`(0) and thus returns a value as argued before. So, we have no issue with $n = 0$ too, and the programs thus must terminate.

(3) Thus, the execution eventually reaches the base cases of `A` and `B` and thus terminates. Since the value is roughly half each time, the time complexity is bounded by $O(\log n)$, and the space complexity (with the defer operations of additions) is also $O(\log n)$.

C. Provide an iterative implementation of `A` (using `while` loop) with an input n without calling any other functions (such as `B`). Note that the output value from your program MUST be the same as that given by `A` for the same input n. [3 marks]

```
def A_itr ( n ):

    if n==0 or n == 1: # just need to handle either one, don't need to do both
        return 1
    result = 0
    indicator = True
    while (n > 1):
        result += n
        if indicator:
            n += 1
        else:
            n = n // 2
        indicator = not indicator
    return result+1
```

D. Suppose we now want to use the functions `A` and `B` for tuples. We amended them to become `A_tuple` and `B_tuple` as follows:

```
def A_tuple(t):
    if len(t) == 1:
        return t[0]
    else:
        return (lambda t: t[0] if len(t)>0 else 0)(t) + B_tuple(t + (len(t),))

def B_tuple(t):
    if len(t) == 1:
        return t[0]
    else:
        return (lambda t: t[0] if len(t)>0 else 0)(t) + A_tuple(t[0:len(t)//2])
```

Provide the time and space complexity of the function `A_tuple` in terms of the length n of the input tuple. Justify your answer. [3 marks]

Time: $O(n)$
Space: $O(n)$
Justification (for time and for space): You should argue about the increment by 1 to the length of tuple by `B_tuple` does not alter the fact that each time `A_tuple` is in turn called again, the value `n` (length of the tuple) is at most half plus 1. This is the same for `B_tuple` when the function execution returns to `B_tuple` the value of `n` (length of the tuple) is at most half plus 1 too. Thus, the execution eventually reaches the base cases of `A_tuple` and `B_tuple` and thus terminates (note that the consideration of tuple of length zero is the same as in Part(B)). Since the value is roughly half each time, the number of times to go through `A_tuple` is $O(\log n)$ and each time it takes time proportional to the length of the input tuple. So the total time is: $n + n/2 + n/4 + \ldots$ which is still $O(n)$. And, the space is also accumulated in the same fashion and thus $O(n)$ too. (Note: as the space is already linear, the tail recursion can save a bit but the complexity remains at linear.)

Question 4: Higher Order Functions [12 marks]

You will be working with the following higher-order function `sumT` which is doing sum of the elements of a tuple `t` with `term(t)`:

```
def sumT(t, term, next):
    if t == ():
        return ()
    else:
        return term(t) + sumT( next(t), term, next )
```

In the following, we are going to use `sumT` to help to perform operations with just one tuple, two tuples, three tuples and tuple of tuples.

A. The function `prefix_sum` takes an input tuple `t` (of numbers) to compute the prefix sum of each term in `t`. That is, the output is another tuple whose i^{th} element is the sum of all elements of `t` with index 0 till index i:

Example execution:

```
>>> prefix_sum( (1,2,3,4,5,6,7,8,9) )
(1, 3, 6, 10, 15, 21, 28, 36, 45)

>>> prefix_sum( (1,2,3,4,5,6,7,8,9,10) )
(1, 3, 6, 10, 15, 21, 28, 36, 45, 55)
```

We can define `prefix_sum` with `sumT` as follows:

```
def prefix_sum( t ):
    return sumT( t, <T1>, <T2> )
```

Please provide possible lambda functions for `<T1>` and `<T2>`. [2 marks]

<T1>: [1 marks]

```
lambda t: (t[0],)
```

Note: this turns out to be a hard question to many. It is not acceptable to just add 1 to n only (as in the given example).

<T2>: [1 marks]

```
lambda t: () if len(t)<2 else (t[0]+t[1],) + t[2:]
```

B. The function `interleave` takes two tuples as input to output a tuple whose elements are alternately taken from the two tuples (while maintaining the same order as in the input tuples). We assume the two input tuples are of the same length.

Example execution:

```
>>> interleave( (1,2,3), ("a", "b", "c") )
(1, 'a', 2, 'b', 3, 'c')

>>> interleave( (1,2,3,4), ("a", "b", "c", (1,2) ) )
(1, 'a', 2, 'b', 3, 'c', 4, (1, 2))
```

We can define `interleave` with `sumT` as follows:

```
def interleave(t1, t2):
      return sumT( t1 + t2, <T3>, <T4> )
```

Please provide possible lambda functions for `<T3>` and `<T4>`. [2 marks]

`<T3>`: [1 marks]

`lambda t: (t[0],) + (t[len(t)//2],)`

This case is to concatenate `t1` with `t2` to use `sumT`, unlike the `average` question. All lambdas must have only a single input `t`.

`<T4>`: [1 marks]

`lambda t: t[1:len(t)//2] + t[len(t)//2+1:]`

C. The function `average` takes three input tuples (of numbers) to compute the average of the three corresponding elements of the three tuples. We assume the three input tuples are of the same length.

Example execution:

```
>>> average( (3, 4, 5), (1, 2, 3), (2, 9, 4) )
(2.0, 5.0, 4.0)

>>> average( (3, 4, 5, 6), (1, 2, 3, 4), (2, 9, 4, 8))
(2.0, 5.0, 4.0, 6.0)
```

We can define `average` with `sumT` as follows:

```
def average(t1, t2, t3):
      return sumT( (t1, t2, t3), <T5>, <T6> )
```

Please provide possible lambda functions for `<T5>` and `<T6>`. [2 marks]

`<T5>`: [1 marks]

`lambda t: ((t[0][0] + t[1][0] + t[2][0])/3,)`

This case is to put `t1`, `t2`, `t3` into a tuple rather than that in the `interleave` question. All lambdas must have only a single input `t`.

`<T6>`: [1 marks]

`lambda t:(t[0][1:],t[1][1:],t[2][1:]) if len(t[0])>1 else ()`

D. We have the following `map` function in our lecture.

```
def map(f, t):
    if t == ():
        return ()
    else:
        return (f(t[0]),) + map(f, t[1:])
```

We can also define `map` with `sumT` as follows:

```
def map(f, t):
      return sumT( t, lambda t: (f(t[0]),), lambda t: t[1:] )
```

For this question, we assume we use the `map` given here in terms of `sumT`. We can now compute the `transpose` for a matrix M given as a tuple of tuples. We have $M[0]$ as the row 1 of the matrix, and $M[1]$ of row 2, and, in general, $M[i]$ as the row i of the matrix. And, $M[i][j]$ is the element at row i and column j. The transpose of a matrix M with n rows and m columns is another matrix M^T with m rows and n columns with its element $M^T[j][i] = M[i][j]$.

Example execution:

```
>>> # below is a matrix M with 3 rows and 3 columns where
>>> # the first row is (1,2,3), second (4,5,6), and third (7,8,9)
>>> # the first column is (1,4,7), second (2,5,8), and third (3,6,9)
>>> M = ( (1,2,3), (4,5,6), (7,8,9) )
>>> transpose(M)
((1, 4, 7), (2, 5, 8), (3, 6, 9))
>>> # the transpose of M has 3 rows and 3 columns,
>>> # with the first row of the transpose taken from M's first column, etc.

>>> N = ( (1,2,3,4), (5,6,7,8), (9,10,11,12) ) # has 3 rows and 4 columns
>>> transpose(N)
((1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12))
>>> # the transpose of N has 4 rows and 3 columns
>>> # with the first row of the transpose taken from N's first column, etc.
```

We can also define `transpose` with `sumT` as follows:

```
def transpose ( M ):
      return sumT( M, <T7>, <T8> )
```

Please provide possible lambda functions for `<T7>` and `<T8>` (that involves `map` that uses `sumT` as given here and not the one in the lecture note). [2 marks]

<T7>:
[1 marks]

```
lambda N: (map(lambda row: row[0], N),)
```

The question asks for the use of `map` in the answer.

<T8>:
[1 marks]

```
lambda N: map(lambda row: row[1:], N) if len(N[0])>1 else ()
```

E. Among the above solutions of yours for `prefix_sum`, `interleave`, and `average`, pick any two of them to discuss their time and space complexity. Note that your answers given in the previous parts must be almost correct before we can grade your given time and space complexity here. So, please choose carefully the two that you are most confident of their correctness to fill in their corresponding two boxes (and leave one box empty). Note that if all 3 boxes are filled, we will grade just the first two. For each box, please state first the time and space complexity, and then provide your reasoning - note that all three may seem to be the same, you still need to point our their specific details (or subtle differences) to receive full credit. [2 marks]

`prefix_sum:`

The time complexity is $O(n^2)$, and space complexity is $O(n^2)$. The function `sumT` is call recursively but each time the length of the tuple is reduced by 1. So the total number of times to called `sumT` is $O(n)$, and in each call, the time needed is the `term(t)` operation (which is a constant) plus the length of the input tuple to create a new tuple (as tuple is immutable) for the next call. So, total time is the usual sum of 1 till n, and is thus $O(n^2)$. As for the space, we need to keep track of tuples of length n, then of length $n-1$ and so on into the recursions. Thus the space complexity is also $O(n^2)$.
We accept the $O(n)$ space complexity if you mentioned the re-use of the space because of the tail recursion.
Note that a correct statement of complexity with no matching correct answer to `prefix_sum` in Part A can receive at most 0.5 score.

`interleave:`

The time complexity is $O(n^2)$, and space complexity is $O(n^2)$.

The solution first concatenates the two input tuples into one tuple, and then call `sumT` with length of $2n$. The recursion is as in the `prefix_sum` case that there are n depth and each call takes time proportional to the (two times the) length of the input tuple (which is $2n$ at the beginning). So, the analysis is very much like that in `prefix_sum` with the difference of a constant 2 and the difference of `term(t)` which takes time linear to the length of the input tuple. Both of these do not change the total time needed for an iteration to be still $O(n)$, and the total time remains the same as that in `prefix_sum`. As for space, we need to keep track of tuples of length $2n$, then length of $2(n-1)$, and so on into the recursions. Thus, the space complexity is also $O(n^2)$.
We accept the $O(n)$ space complexity if you mentioned the re-use of the space because of the tail recursion.
Note that a correct statement of complexity with no matching correct answer to `interleave` in Part B can receive at most 0.5 score.

`average:`

The time complexity is $O(n^2)$, and space complexity is $O(n^2)$.
This case is almost the same as the case of `prefix_sum`, with a difference in the constant of 3. We accept the $O(n)$ space complexity if you mentioned the re-use of the space because of the tail recursion. Note that a correct statement of complexity with no matching correct answer to `average` in Part C can receive at most 0.5 score.

F. Analyse the time and space complexity of `transpose` in terms of n and m where n is the number of rows and m is the number of columns of the input matrix. Note that your answers given to `transpose` in the above must be almost correct before we can grade your attempt here. [2 marks]

`transpose:`

Time: $O(n^2m^2)$

Space: $O(\max(nm^2, n^2m))$ (or $O(nm)$ if we consider tail recursion.)

Justification (for time and for space):

This is hard for ALL of you (and myself too) - we need to be very careful and to break down each piece to (hopefully) get the right "counting".

There are 2 levels to this: first <T7> and <T8> are recursive calls (in one level by themselves using `map`) and we need to understand their complexity first. Then we can understand them in the next level used by `simT` (in `transpose`).

So, first level first. For a given matrix of size $n \times m$, <T7> is to go through each tuple (as a row) within a tuple (of all rows) to get its first element, and then recursively doing this with a (new) tuple with 1 less row. So, the time needed here is $O(1)$ to get the first element, plus $O((n-1)m)$ to create the new tuple to go into the recursion (Note: actually this can just be $O(n-1)$ due to shallow copying, but we keep it as $O((n-1)m)$ because <T8> needs it). Then, another $O(1)$ to get the first element again, plus $O((n-2)m)$ to create the new tuple to go into the next recursion, and so on. So, with n number of recursions, the total time needed is: $O((n-1)m) + O((n-2)m) + O((n-3)m) + +O((n-n)m) = O(n^2m)$. Similarly for <T8>, instead of getting the first element in <T7>, it creates one row with 1 less column in each of the n levels of recursion to construct a matrix of size $n \times (m-1)$ for the recursion into `sumT` (in `transpose`).

Next, <T7> and <T8> are used in the recursion of the next (normal) level of `sumT` (in `transpose`). In here, the size of matrices for <T7> and <T8> are initially $n \times m$, then $n \times (m-1)$ (one less column), then $n \times (m-2)$ (two less columns), and so on. So we plug into the previous paragraph (the decreasing sizes of matrices) to get the time complexity of: $O(n^2m) + O(n^2(m-1)) + \cdots + O(n^2(m-m))$, which is $O(n^2m^2)$, for the m recursions due to m columns.

As for space complexity, for the first level of <T7> and <T8> using `map`, the largest requirement is $O((n-1)m) + O((n-2)m) + \cdots O((n-n)m) = O(n^2m)$. (This can actually be $O(nm)$ if we consider <T7> and <T8> as tail recursion and only the needed matrix of size $n \times (m-1)$ is stored.) For the second level of <T7> and <T8> in `sumT`, the space is then to stack up the original matrix with $O(nm)$ space, then matrix of $O(n(m-1))$ space (of 1 less column), and so on to have $O(nm^2)$ in total. (Again, this can actually be still $O(nm)$ if we recognize `sumT` as a tail recursion.)

— END OF PAPER —