CS1010X — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

26 March 2022 **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**. DO NOT WRITE YOUR NAME ON THE QUESTION PAPER!
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 **FIVE (5) questions** and **NINE (9) 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 this question paper. If you write something important as your answer outside the space provided, please make clear with some marking. At the end of the test, please submit your question paper only. No extra sheets will be accepted as answers.
7. You are allowed to use pencils. Please be sure your handwriting is legible and neat, and you have proper indentation as needed for your Python codes to avoid misinterpretation.

GOOD LUCK!

Question	Marks
Q1	
Q2	
Q3	
Q4	
Q5	
Total	

Question 1: Warming Up [5 marks]

A. For the following Python code (note that nested-if condition in a lambda function is valid still), write the output of the `print` statement. [2 marks]

```
z = lambda x: 1 if (x < 0 or x > 10) else (2 if x%2==0 else 3)
def f(a, b):
    sum = 0
    for i in range(a, b):
        sum += z(i)
    return sum

print(f(-6,16))
```

38

B. For the following Python code, write the outputs (3 lines in total) of the `print` statements. [3 marks]

```
t = (1,2,3,4)
lst = [66, 77]
def gn(c):
    c[0] = [88]
    c[1][1] = 55
    return
def fn( a, b ):
    m = [ a, b ]
    a = b
    gn(m)
    return m

r = fn(t, lst)
print(t)
print(lst)
print(r)
```

(1, 2, 3, 4)
[66, 55]
[[88], [66, 55]]

Question 2: Something familiar first [4 marks]

```
def compose(f, g):
    return lambda x:f(g(x))

def twice(f):
    return compose(f, f)

def thrice(f):
    return compose(f, compose(f, f))

def repeated(f, n):
    if n == 0:
        return identity
    else:
        return compose(f, repeated(f, n - 1))

add1 = lambda x: x + 1
identity = lambda x: x
```

Given the above familiar codes, you are to write out the output of each given `print` statement. Each part is 1 mark.

```
print(twice(thrice)(add1)(1))
```

10. Applied two times to applying three times to a function (add1 in this case). So, the result is to applying 9 times of add1 to the number 1.

```
print(twice(thrice)(twice)(add1)(1))
```

513. Same logic as the previous part: so applied 2^{3^2} times of add1 to the number 1.

```
print(thrice(twice)(thrice(add1))(1))
```

25. Slightly different from the previous parts because of the parentheses: applied 2^3 to the function "thrice(add1)" (which is to add 3 to the input). That is, applied 8 times of add 3 to the number 1.

```
print(twice(thrice)(repeated(thrice(add1),100))(1))
```

2701
This is a real test on whether you can interpret the semantic of the statement, rather than doing the naive tracing. The back part of the "repeated(thrice(add1))" says that repeats 100 times the given function "thrice(add1)" (which is to add 3 to whatever given number/input), and it thus means, on the whole, "to add 300 to whatever given number" (in our case "1"). The front part "twice(thrice)" says that "applies 3 times to whatever function (at the back, "which is to add 300 to whatever given number"), and then do this again (because of "twice") and resulted in "applying 9 times to whatever function" (at the back, "which is to add 300 to whatever given number"). So, the whole statement is to add 2700 to whatever given number, which is "1", and we thus have 2701 as the answer. Similar arguments are used in previous parts.

Question 3: Not really unfamiliar [10 marks]

A. You are given the following python code that works with input of two tuples `s` and `t`, of the same length.

```
def unknown(s, t):
    if len(s) != len(t):
        return None
    else:
        j = 1
        result = list(s)
        while j < len(s):
            for i in range(len(s)-1, j-1, -1):
                if t[i] == t[i-j]:
                    result[i] = result[i] + result[i - j]
            j *= 2
        return result
```

Suppose we run the above code with the following 3 statements:

```
t = (0, 0, 0, 0, 1, 1, 1, 2, 3, 3, 4, 4, 4, 4, 4, 5, 5)
s = (1, 2, 1, 4, 2, 3, 4, 4, 2, 3, 4, 4, 4, 4, 4, 9, 10)
print(unknown(s, t))
```

What is the output from the `print` statement? [3 marks]

[1, 3, 4, 8, 2, 5, 9, 4, 2, 5, 4, 8, 12, 16, 20, 9, 19]

Explanation (if any to assist with understanding your thinking about the purpose of the code - good for receiving partial credit if your answer may not be completely correct): This is actually a segmented prefix sum where the tuple t indicates the segment of the corresponding item in s.

What is the time and space complexity of the `unknown` function in terms of the length n of the tuple `s` and `t`. (Note that `t` is always in the form that `t[i]` $\leq$ `t[j]` for `i < j`. This information is not needed here for your analysis but just in case you have query on `t`.) A correct but too big an upper bound will not receive any credit. [3 marks]

Time: $O(n \log n)$
Space: $O(n)$

Justification (for time and for space): This is very much like the parallel prefix sum too where the looping is done $\log n$ time where n is the length of the input tuples, and each loop requires O(n) time. So, the stated time complexity. As for the space complexity, there is a resulting tuple needed and thus O(n) space.

B.

```
import math
def puzzle(n):
    sum = 0
    for i in range(1, n):
        for j in range(1, i*n):
            for k in range(1, int(math.log(i))):
                for m in range(1, k):
                    if m > 1000000:
                        break
                    sum = sum + 1
    return sum
```

What is the time and space complexity of the `puzzle` function in terms of the input n. A correct but too big an upper bound will not receive any credit. [4 marks]

Time: $O(n^3 \log n)$
Space: $O(1)$

Justification (for time and for space):

This is a tricky one. First notice that the innermost m-loop is of a constant time. This is because once m hits over 1000000 the loop will stop and thus it is never a function of the input n, and we can thus treat it as O(1) for the time analysis and thus ignore it is a "loop". As a result, we now have just 3 nested looping, i, j and k. Then, for a given i, the second loop of j will do $i \cdot (n-1)$ times to the k-loop of $\log i$ times. So, the sum of all these for i from 1 to n is: $1(n-1) + 2(n-1)\log 2 + 3(n-1)\log 3 + ... + (n-1)(n-1)\log(n-1)$, which is $(n-1)(1 + 2\log 2 + 3\log 3 + ... + (n-1)\log(n-1))$. Then by setting all $\log i$ to a larger number $\log n$ to move it outside of the parentheses, we have $(n-1)(\log n)(1+2+3+...(n-1))$, which is now familar to you as O($n(\log n)n^2$), and is thus O($n^3 \log n$). As for the space complexity, the program does not use anything more than just a few simple variables of constant space.

Question 4: Counting Squares in Grid [4 marks]

Given a square grid of size n by n, we want to write a program to count the total number of squares of size 1 by 1 (there are $n \times n$ of them), size 2 by 2 (depending on n), and so on until size of n by n (just 1 such big square). This is in the same spirit as the familiar House of Cards problem we know: instead of counting triangles in House of Cards, we are counting squares here.

For a 1 by 1 grid, there is just 1 square in the grid.

For a 2 by 2 grid, there are 4 squares of size 1 by 1, and 1 square of size 2 by 2, for a total of 5 squares in the grid.

For a 3 by 3 grid, there are 9 squares of size 1 by 1, 4 squares of size 2 by 2, and 1 square of size 3 by 3, for a total of 14 squares in the grid.

For a 4 by 4 grid, what is the total number of squares we can find? [1 marks]

30

Now you can write a recursive program to do the count on the total number of squares for a grid of size n by n. It is possible that there might be a closed form formula here to state the total number of squares, but you are not supposed to use any formula but simple recursive thinking to solve this question. Do provide simple explanation of your recursive thinking too. [3 marks]

```
def num_sq_given_grid_of(n):

    if n==1:
        return 1
    else:
        undercounted = 0
        for i in range(1, n+1):
            undercounted = undercounted + 2*(n-i)+1
        return undercounted + num_sq_given_a_grid_of(n-1)
```

The undercounted part captures those not included in the smaller size problem of $n-1$. It simply loops through from 1 to n to collect those missing $2(n-1)+1$ squares of size 1x1, and then $2(n-2)+1$ squares of size 2x2, and so on till $2(n-n)+1=1$ of size nxn.

Question 5: Higher Order Functions [9 marks]

You will be working with the following higher-order functions `sumT` and `foldT` which are similar to the `sum` and `fold` as given in our lessons but now working on tuples. (To simplify our questions to produce numbers, we set the base cases in both functions to return 0 instead of their usual tuple or `f(0)`.)

```
def sumT(t, term, next):
    if t == ():
        return 0
    else:
        return term(t) + sumT( next(t), term, next)

def foldT(t, op, f):
    if t == ():
        return 0
    else:
        return op( f(t), foldT( t[:-1], op, f) )
```

A. The function `alt_sum_sq` takes an input tuple `t` (of numbers) to compute the alternating sum of the square of each term in `t`. That is, the output is:

$$\sum_{i=0}^{len(\mathtt{t})-1} (\mathtt{t}[i])^2 \cdot (-1)^i$$

Example execution:

```
>>> alt_sum_sq( (6,7,8) )   # = 6*6 - 7*7 + 8*8
51

>>> alt_sum_sq( (6,7,8,9) )   # = 6*6 - 7*7 + 8*8 - 9*9
-30
```

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

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

Please provide possible lambda functions for `<T1>` and `<T2>`. We reserve the right to give no credit if your `<T1>` and `<T2>` do not match up to give a correct thinking to solve the problem. For example, no credit could be given when either `<T1>` or `<T2>` is left unanswered. [3 marks]

`<T1>`: [2 marks]

```
lambda t: t[0]**2 if len(t)==1 else t[0]**2-t[1]**2
```

`<T2>`: [1 marks]

```
t: t[2:]
```

We can also define `alt_sum_sq` with `foldT` as follows:

```
def alt_sum_sq( t ):
    return foldT( t, <T1>, <T2> )
```

Please provide possible lambda functions for `<T1>` and `<T2>`. As in the above, we reserve the right to give no credit if your `<T1>` and `<T2>` do not match up to give a correct thinking to solve the problem. [3 marks]

`<T1>`: [1 marks]

```
lambda x,y: x + y
```

`<T2>`: [2 marks]

```
lambda t: -t[-1]**2 if len(t)%2==0 else t[-1]**2
```

B. Given two tuples of the same length, `t` and `p`, suppose we want to do the computation as follows:

```
def position_sum(t, p):
    result = 0
    for i in range(0, len(t)):
        result = result + t[i]*p[i]
    return result
```

Instead of the above simple code, we want to exercise our skill working with higher order functions to achieve the same computation. We need to do this in two parts where the first is a `helper` function to process the input `t` and `p` into a suitable format that you can use in `sumT` and `foldT` but you are NOT to do the calculation (this means multiplication) belonging to `sumT` and `foldT`. If you have trouble with writing this function, you should at least provide description on what your `helper` function is supposed to produce so that your solution in the next part can make some sense still. [1 marks]

```
def helper ( t, p ):
#
# IMPORTANT: You are to write simple iterative or recursive program WITHOUT
#   the use of any other Python built-in functions (eg. accumulate, map, etc)
#   You may use len function to check the length of a tuple. Also note that
#   helper function is to arrange the items of t and p in some way without
#   doing multiplication belonging to sumT and foldT.

    result = ()
    for i in range(len(t)):
        result += ((t[i], p[i]),)
    return result
```

This is just one of many ways to do this. The simplest one is just "return t + p" (and with the corresponding different answers to the next part that you can figure out).

The next step is the actual use of `sumT` or `foldT`. You can choose to do with either one. Please **DO NOT ANSWER BOTH** as we will only mark the first non-empty version if you do both.

The version with `sumT` is as follows:

```
def position_sum( t, p ):
    return sumT ( helper(t, p), <T1>, <T2> )
```

Please provide possible lambda functions for `<T1>` and `<T2>`. As before, we reserve the right to give no credit if your `<T1>` and `<T2>` plus the `helper` function do not match up to give a correct thinking to solve the problem. [2 marks]

`<T1>`: [1 marks]

```
lambda t: t[0][0] * t[0][1]
```

`<T2>`: [1 marks]

```
lambda t: t[1:]
```

If you have done the above with `sumT` for `position_sum`, please ignore the following with `foldT`. Otherwise, please proceed to do the following with `foldT`:

```
def position_sum( t, p ):
    return foldT( helper(t, p), <T1>, <T2> )
```

Please provide possible lambda functions for `<T1>` and `<T2>`. As before, we reserve the right to give no credit if your `<T1>` and `<T2>` plus the `helper` function do not match up to give a correct thinking to solve the problem. [2 marks]

`<T1>`: [1 marks]

```
lambda x,y: x+y
```

`<T2>`: [1 marks]

```
lambda t: t[-1][0]*t[-1][1]
```

— END OF PAPER —