CS1010S — Programming Methodology
School of Computing
National University of Singapore

Re-Midterm Test

14 October 2016 **Time allowed:** 1 hour 45 minutes

Student No:	**S**	**O**	**L**	**U**	**T**	**I**	**O**	**N**	**S**

Instructions (please read carefully):

1. Write down your matriculation number on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is **an open-sheet test**. You are allowed to bring one A4 sheet of notes (written on both sides).
3. This paper comprises **FOUR (4) questions** and **EIGHTEEN (18) pages**. The time allowed for solving this test is **1 hour 45 minutes**.
4. The maximum score of this test is **100 marks**. Students attempting the midterm test for the second time is subject to a **maximum score of 60 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the answer sheet; no extra sheets will be accepted as answers.
7. The pages marked "scratch paper" in the question set may be used as scratch paper.
8. You are allowed to detach the sheets while you solve the questions. Please make sure you staple them back in the right order at the end of the test.
9. You are allowed to use pencils, ball-pens or any other writing instrument as you like, as long as it is not red in colour.

GOOD LUCK!

Question	Marks	Remark
Q1		
Q2		
Q3		
Q4		
Total		

Question 1: Python Expressions [30 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 the response printed by the interpreter for the final expression entered and **write the exact output in the answer box**. If the interpreter produces an error message, or enters an infinite loop, explain why. You may show your workings **outside the answer box** in the space beside the code. Partial marks will be awarded for workings if the final answer is wrong.

A. [5 marks]

```
s = "Happy Re-midterm!"
while s:
    out = s[0]
    s = s[1:-1]
    if out == " ":
        break
print(s)
```

```
'Re-mi'
```

B. [5 marks]

```
a = (1, 2)
b = (a, 4, 5)
c = b + ()
print(b is c, b[0] is c[0], a in b, a in c)
```

```
False True True True
```

C. [5 marks]

```
x = 7
y = 5
def g(y, z):
    x = z * y
    return f(x)
def f(x):
    return y + x
print(g(x, y))
```

```
40
```

D. [5 marks]

```
n = (3, 2, 1)
m = (1, 2, 3)
if n < m:
    n = m + n
else:
    m = n + m
if n > m:
    n = (m, n)
else:
    m = (n, m)
print(n)
print(m)
```

```
(3, 2, 1)
((3, 2, 1), (3, 2, 1, 1, 2, 3))
```

E. [5 marks]

```
j = 6
for i in range(j, -j, -1):
    if i % 2 == 0:
        j += 1
        i -= 1
    else:
        j = j // 2
    if i == j:
        break
print(j)
```

```
1
```

F. [5 marks]

```
boo = lambda x: lambda y: bar(y)
bar = lambda x: lambda y: x(y)
print(boo(int)(tuple)('123'))
```

```
('1', '2', '3')
```

Question 2: Linear 2048 [28 marks]

INSTRUCTION: You are not allowed to use any higher-order functions for this question.

The game 2048 is played on a 4x4 grid. We simplify the game by playing it on a single line of numbers, which is represented by using a tuple, e.g., `(0, 2, 0, 2, 4, 0, 0, 16, 8)`. The length of the line for a given game is not fixed.

A. [Warm up] Write a function `pad_right` that takes as input a line of numbers l, represented as a tuple and a length n, which is an integer. The function will return a new tuple, which is l appended with 0s to the end of the tuple such that the length of this new tuple is n.

Example:

```
>>> pad_right((2, 2, 4, 8), 6)
(2, 2, 4, 8, 0, 0)  # add 0s to make the length 6

>>> pad_right((2, 2, 4, 2, 4, 8), 6)
(2, 2, 4, 2, 4, 8)  # length is already 6, so no need to add 0s
```

[6 marks]

```
def pad_right(line, n):
    while len(line) < n:
        line += (0,)
    return line
```

A move in 2048 will shift all non-zero blocks to one side, and then combine two adjacent blocks of the same value to give a single block. This can be broken into two steps: 1) shifting all non-zero blocks to one size, and 2) combining two adjacent blocks of the same value to a single block.

The function `squash_left` takes as input a tuple of integers, and returns a tuple where every non-zero element of the input is shifted as far left as possible. The length of the output tuple must be equal to the input and any unfilled elements will be padded with 0s.

Example:

```
>>> squash_left((0, 2, 0, 2, 4, 0, 8, 16))
(2, 2, 4, 8, 16, 0, 0, 0)  # all non-zero elements move to the left
```

```
>>> squash_left((16, 0, 0, 0, 4, 0, 16, 0))
(16, 4, 16, 0 , 0 , 0, 0, 0)

>>> squash_left((2, 4, 16, 8, 2, 4))
(2, 4, 16, 8, 2, 4)   # no zero elements to squash
```

B. Provide a **recursive** implementation of the function `squash_left(line)`.

Hint: Think of it as sending 0s to the right. [6 marks]

```
def squash_left(line):
    if line == ():
        return ()
    elif line[0] != 0:
        return (line[0],) + squash_left(line[1:])
    else:
        return squash_left(line[1:]) + (0,)
```

C. What is the order of growth in terms of time and space for the function you wrote in Part (B). Briefly explain your answer. [2 marks]

Time: $O(n^2)$ where n is the length of line. This is because there are n recursive calls and in each call there is a tuple concatenation and slicing which takes n time.

Space: $O(n^2)$ where n is the length of line. This is because there are n recursions, taking up space on the stack and in each recursion, a new tuple of n length is created from the slicing.

Other answers are acceptable according to what was written in part C. If the code does not make sense, then we cannot compute an order of growth, so no marks will be given.

D. Provide an iterative implementation of the function `squash_left(line)`.

Hint: Think of it as simply extracting the non-zero elements and padding the right with 0s to the original length. [6 marks]

```
def squash_left(line):
    new_line = ()
    for i in line:
        if i != 0:
            new_line += (i,)
    return pad_right(new_line, len(line))
```

E. What is the order of growth in terms of time and space for the function you wrote in Part (D). Briefly explain your answer. [2 marks]

Time: $O(n^2)$ where n is the length of line. This is because while the for-loop loops n times, each loop creates a new tuple of n length by concatenation.

Space: $O(n)$, because a new tuple of length n being created every iteration.

Other answers are acceptable according to what was written in part E. If the code does not make sense, then we cannot compute an order of growth, so no marks will be given.

After shifting all the non-zero elements to the left, the next step is to combine the two left-most adjacent elements with the same value into one element that is double in value. All the elements of the right of these two elements will be shifted to the left. The function `combine` takes in a tuple of numbers and returns a tuple after performing this combining action.

Example with the elements that will be combined underlined:

```
>>> combine((2, 2, 4, 4, 8, 8, 16))
(4, 8, 16, 16, 0, 0, 0)  # the 2s combine to become 4
                            # the 4s combine to become 8
                            # the 8s combine to become 16
                            # 16 cannot combine so it is left

>>> combine((2, 2, 2, 4, 2, 2, 2, 8, 0, 0))
(4, 2, 4, 4, 2, 8, 0, 0, 0, 0)  # the left most 2s are combined

>>> combine((2, 2, 2, 2, 2, 2, 2))
(4, 4, 4, 2, 0, 0, 0)
```

F. Provide an implementation for the function `combine(line)`. You may assume the line has already been squash-left, i.e., only contains trailing 0s.

Hint: It might be more straightforward to implement using recursion. [6 marks]

Recursive solution:

```
def combine(line):
    if len(line) <= 1:
        return line
    elif line[0] == line[1]:
        return (line[0]*2,) + combine(line[2:]) + (0,)
    else:
        return (line[0],) + combine(line[1:])
```

Iterative solution:

```
def combine(line):
    new_line = ()
    i = 0
    while i < len(line):
        if i < len(line)-1 and line[i] == line[i+1]:
            new_line += (line[i]*2,)
            i += 2
        else:
            new_line += (line[i],)
            i += 1
    return new_line
```

Question 3: Higher-Order 2048 [18 marks]

INSTRUCTION: This question should be solved using higher-order function, and not by recursion, iteration or reusing the functions defined in Question 2, unless stated otherwise.

A. The function `squash_left` from Question 2B can be expressed in terms of a higher-order function listed in the Appendix together with the `pad_right` function from Question 2A. Provide the missing code in the given implementation.

Hint: What are you actually doing with the 0s? [6 marks]

```
def squash_left(line):
    return pad_right(
            filter(lambda x:x != 0, line))
            , len(line))
```

B. Our linear 2048 game is considered over when there are no 0s in the line and no two adjacent elements are equal, i.e., squash-left and combine will produce the exact same result. The function `game_over` takes as input a line of the game (which is a tuple of integers) and returns `True` if the game is over, and `False` otherwise.

Example:

```
>>> game_over((2, 4, 8, 4, 4, 2))
False  # still have adjacent 4s

>>> game_over((2, 4, 8, 4, 8, 2))
True
```

The function `game_over` can be expressed using the higher-order function `fold2` as follows:

```
def game_over(line):
    <PRE>
    if 0 in line:
        return False
    else:
        return fold2(<T1>, <T2>, <T3>, <T4>, <T5>, <T6>)
```

Please provide possible implementations for the terms `T1`, `T2`, `T3`, `T4`, `T5` and `T6`. You may also optionally define functions in `<PRE>` if needed. [6 marks]

*optional <PRE>:	
<T1>:	`lambda x, y: x and y`
<T2>:	`lambda n: line[n]`= line[n+1]!
<T3>:	`0`
<T4>:	`lambda n: n+1`
<T5>:	`len(line)-2`
<T6>:	`True`

C. The function `combine` from Question 2F can be expressed in terms of `fold2` listed in the Appendix as follows:

```
def combine(line):
    <PRE>
    return pad_right(
               fold2(<T1>, <T2>, <T3>, <T4>, <T5>, <T6>),
               len(line))
```

Please provide possible implementations for the terms `T1`, `T2`, `T3`, `T4`, `T5` and `T6`. You may optionally define functions in `<PRE>` if needed. [6 marks]

*optional `<PRE>`:

```
    def term(n):
        if n < len(line)-1 and line[n] == line[n+1]:
            return (line[n]*2,)
        else:
            return (line[n],)

    def next(n):
        if n < len(line)-1 and line[n] == line[n+1]:
            return n+2
        else:
            return n+1
```

`<T1>:`

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

`<T2>:`

```
term
```

`<T3>:`

```
0
```

`<T4>:`

```
next
```

`<T5>:`

```
len(line)-1
```

`<T6>:`

```
()
```

Question 4: Uber Driving [24 marks]

Warning: Please read the entire question clearly before you attempt this problem!!

An Uber driver has to keep a record of trips that he has driven. A trip contains the following three information: 1) name of the passenger, 2) the pick-up location, and 3) the drop-off location. Each of these information are strings.

The function `create_trip` will take these 3 information as inputs, and return a trip object.

The functions `get_name`, `pick_up` and `drop_off` each takes a trip as input, and returns the name, pick-up location and drop-off location respectively.

A. State and describe the data structure you will use and to implement a trip. [2 marks]

A trip will be a tuple of three elements: name of passenger, pick-up and drop-off location.

B. Provide an implementation for the functions `create_trip`, `get_name`, `pick_up` and `drop_off`. [4 marks]

```
def create_trip(name, src, dest):
    return (name, src, dest)



def get_name(trip):
    return trip[0]



def pick_up(trip):
    return trip[1]



def drop_off(trip):
    return trip[2]
```

[Important!] For the remaining parts of this question, **you should not break the abstraction of a trip.**

The record of trips is kept in a log in the sequence trips are taken. The functions `empty_log` returns an empty log and `add_trip` takes as inputs a log and a trip and returns a log with the trip added to it.

C. State and describe the data structure you will use and to implement the log. [2 marks]

A log is simply a tuple of trips in order. One possible enhancement is to insert "empty" trips with no passenger to link up the drop-off and pick-up locations between fares.

D. Provide an implementation for the functions `empty_log` and `add_trip`. [4 marks]

```
def empty_log():
    return ()




def add_trip(log, trip):
    log += (trip,)
    return log

# with the enhancement
def add_trip(log, trip):
    if get_dest(log[-1]) != get_src(trip):
        log += (create_trip("", get_dest(log[-1]), get_src(trip)),)
    log += (trip,)
    return log
```

Assume that the function `cost(a, b)` returns the price that the passenger will pay for going from location *a* to *b*.

E. Implement a function `total_earning` which takes as input a log, and return the total earnings of all the trips in the log. [4 marks]

```
def total_earnings(log):
    return sum(cost(pick_up(x), drop_off(x)) for x in log)

# add a filter if using the enhancement
if get_name(x) == ""
```

F. Assume that the function `time(a, b)` returns the time taken to drive from location *a* to *b*. The function `total_time` takes as input a log, and returns the total time the driver has been driving for.

Note that if the adjacent trips do not end and start at the same location, you need to account for the time taken to drive to the location of the next trip. For example, suppose the log has a trip from Jurong to Changi followed immediately by a trip from Bedok to Clementi. Since Changi is not Bedok, you must account for the travel time from Changi to Bedok.

Provide an implementation of `total_time`. [4 marks]

```
def total_time(log):
    last = pick_up(log[0])
    total = 0
    for t in log:
        p, d = pick_up(t), drop_off(t)
        if last != p:
            total += time(last, p)
        total += time(p, d)
        last = d
    return total

# with enhancement
def total_time(log):
    return sum(time(pick_up(x), drop_off(x)) for x in log)
```

G. Write a function `earnings_per_hour` which takes as inputs a log and returns the average earnings obtained per hour. Assume that `cost` returns an amount in dollars (float) and `time` returns a time in minutes (integer). [4 marks]

```
def earnings_per_hour(log):
    return total_earnings(log) / total_time(log) / 60
```

Appendix

The following are some functions that were introduced in class. For your reference, they are reproduced here.

```
def sum(term, a, next, b):
  if (a > b):
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def product(term, a, next, b):
  if a > b:
    return 1
  else:
    return term(a) * product(term, next(a), next, b)

def fold(op, f, n):
  if n==0:
    return f(0)
  else:
    return op(f(n), fold(op, f, n-1))

def fold2(op, term, a, next, b, base):
    if a > b:
        return base
    else:
        return op(term(a), fold2(op, term, next(a), next, b, base))

def enumerate_interval(low, high):
    return tuple(range(low,high+1))

def filter(pred, seq):
    if seq == ():
        return ()
    elif pred(seq[0]):
        return (seq[0],) + filter(pred, seq[1:])
    else:
        return filter(pred, seq[1:])

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

Scratch Paper

Scratch Paper

Scratch Paper

— E N D O F P A P E R —