CS1010S — Programming Methodology
School of Computing
National University of Singapore

Re-Midterm Test Solutions

17 March 2017 **Time allowed:** 1 hour 30 minutes

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 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 **TWENTY-TWO (22) pages**. The time allowed for solving this test is **1 hour 30 minutes**.
4. The maximum score of this test is **75 marks**. Students attempting the re-midterm test for the second time is subject to a **maximum score of 38 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 un-staple 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 fountain pens, as you like as long as it is legible (no red color, please).

GOOD LUCK!

Question	Marks	Remark
Q1	/ 20	
Q2	/ 22	
Q3	/ 10	
Q4	/ 23	
Total	**/ 75**	

This page is intentionally left blank.

It may be used as scratch paper.

Question 1: Python Expressions [20 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.
```
a, b = 7, 13
def f():
    return 100 * a + b
def g(a, b):
    return 1000 * a + b
g(f(), f())
```
[4 marks]

```
713713
```

B.
```
def f(t):
    if t == 0:
        return ('0')
    elif t == (0,):
        return ('(0)')
    elif len(t) == 1:
        return 0
    else:
        return (f(t[1]), t[0])
x = (1, (2, (0, )))
f(x)
```
[4 marks]

```
(('(0)', 2), 1)
```

C.
```
a = lambda x : (lambda y : x(y))
b = lambda x : x * 3
c = lambda x : x + 3
a(b)(c(7))
```
[4 marks]

```
30
```

D.
```
n, x = 1234567890, 0
while n:
    r = n % 10
    n //= 10
    if r == 8:
        continue
    print(r)
    if r in range(5, 9):
        break
print("Done")
```
[4 marks]

```
0
9
7
'Done'
```

E.
```
a = 'abraces'
b = 'abracad'
c = 'abracadabra'
if a < b:
    print('Hai')
    if b < c:
        print('ku')
else:
    print('this')
if b < c:
    print('is')
elif a > b:
    print('not')
```
[4 marks]

```
'this'
'is'
```

This question tests the understanding of string comparisons and if-else statements

Question 2: Mancala [22 marks]

"Mancala is a generic name for a family of 2-player turn-based strategy board games played with small stones or seeds and rows of holes or pits in the earth, a board or other playing surface."
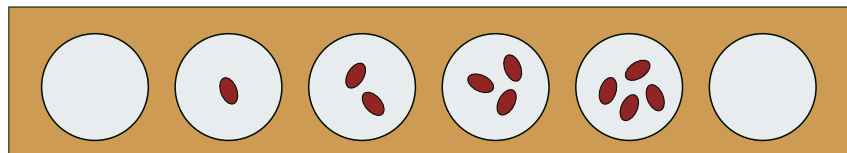
*Source: Wikipedia*

Figure 1: A mancala board

For this question, our mancala board is a row of pits, represented by a string of digits. Each character position in the string represents a pit on the board and each digit represents the number of seeds in the pit.

For example, the string `"012340"` represents a board like this:

Each move in the game is played by "sowing" some number of seeds one at a time into each pit moving from left to right. For example, sowing 3 seeds starting at the second pit using the example board `"012340"` given above will produce a board `"023440"`.

In other words, with 3 seeds in your hand, you drop one seed in each pit starting from the second, until you run out of seeds or reach the end of the board.
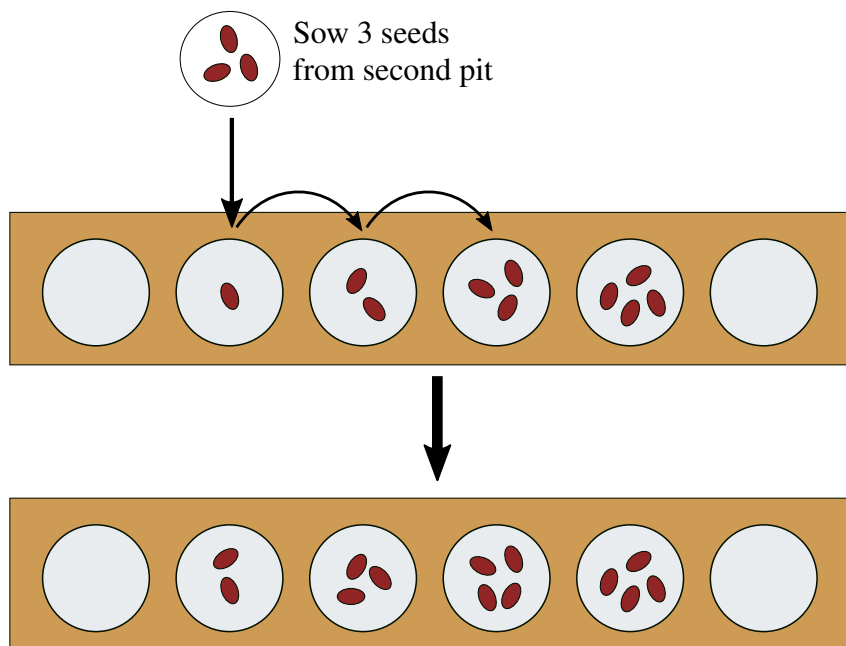
The following game rules will be followed:

- The first pit is numbered 0.

- If a seed is dropped into a pit that contains 9 seeds, all the seeds will "spill over" and 0 seeds will be left in the pit.

- If you reach the end of the board when sowing and still have seeds leftover, the extra seeds will be discarded.

The function `sow` takes as inputs a board (which is a string), a position (which is an integer) and a number of seeds (which is an integer), and returns a board that represents the state after the seeds are sowed.

Examples:

```
>>> sow("012340", 1, 3)  # add 1 seed each starting from index 1
"023440"

>>> sow("999999", 3, 3)  # pits with 9 seeds become 0 when sowed
"999000"

>>> sow("44444", 3, 4)  # extra seeds are discarded
"44455"
```

A. **[Warm up]** To make the task of implementing the `sow` function easier, we will first define a function `add1` which you may then use later. `add1(board, at)` takes as input a board and a position, and returns a board with one seed added to the given position.

Example:

```
>>> add1("012340", 1)  # adds a seed at index 1
"022340"

>>> add1("999999", 3)  # pit overflows and becomes 0
"999099"
```

Provide an implementation of the function `add1`.   [4 marks]

```python
def add1(num, at):
    return = num[:at] + str((int(num[at])+1) % 10) + num[at+1:]
```

B. Provide an **<u>iterative</u>** implementation of the function `sow`. [4 marks]

```python
def sow(board, at, seeds):
    while at < len(board) and seeds > 0:
        board = add1(board, at)
        at += 1
        seeds -= 1
    return board
```

or using for-loop

```python
def sow(board, at, seeds):
    for i in range(seeds):
        if at+i >= len(board):
            break
        board = add1(board, at+i)
    return board
```

C. What is the order of growth in terms of time and space for the function you wrote in Part (B). You may assume that `add1` runs in constant time and space if you have used it in your function. Briefly explain your answer. [3 marks]

Time:$O(seeds)$ because the loop iterates *seeds* number of times.

Space:$O(1)$, because even though there are no delayed operations or new objects being created every iteration.

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

D. Provide a <u>recursive</u> implementation of the function sow.   [4 marks]

```python
def sow(board, at, seeds):
    if at >= len(board) or seeds == 0:
        return board
    else:
        board = add1(board, at)
        return sow(board, at+1, seeds-1)
```

E. What is the order of growth in terms of time and space for the function you wrote in Part (D). You may again assume that `add1` runs in constant time and space if you have used it in your function. Briefly explain your answer.

*Hint: Note the time and space complexity of string slicing.*   [3 marks]

Time:$O(seeds)$ because there are *seeds* recursion calls.

Space:$O(seeds)$ because each of the *seeds* recursion call takes up space in the stack.

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

9

F. Consider this modification to the game rules to allow wrapping when sowing. If you reach the end of the board and there are still seeds left to be sown, you return to the first pit and continue sowing.

For example, if the board is `1234` and 5 seeds are sowed starting from index 1, the outcome will be as illustrated:
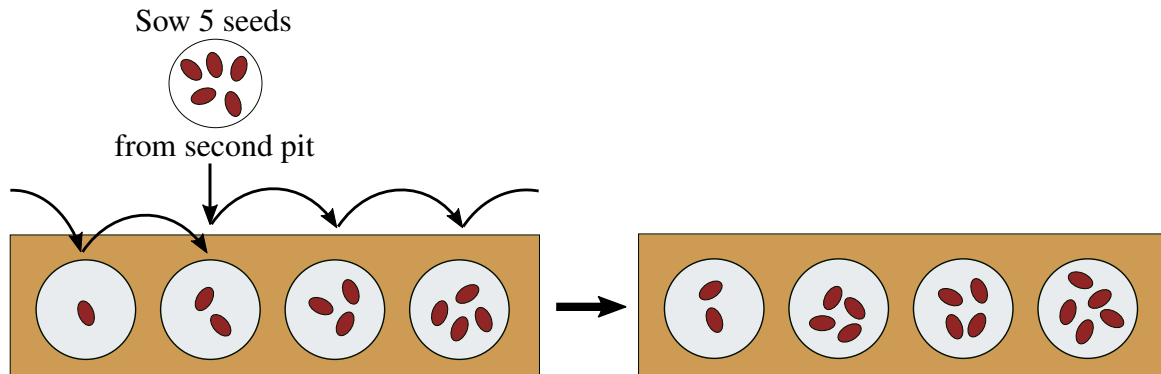
The function `wrapping_sow` works the same as the function `sow` with this new wrapping rule.

Example:

```
>>> wrapping_sow("1234", 1, 5)
"2445"

>>> wrapping_sow("9987", 2, 7)  # Can wrap around multiple times
"1009"
```

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

```python
def wrapping_sow(board, at, seeds):
    for i in range(seeds):
        board = add1(board, at)
        at = (at + 1) % len(board)
    return board
```

or with recursion

```python
def wrapping_sow(board, at, seeds):
    if seeds == 0:
        return board
    elif at >= len(board):
        return wrapping_sow(board, 0, seeds)
    else:
        board = add1(board, at)
        return wrapping_sow(board, at+1, seeds-1)
```

Question 3: Higher-Order Mancala [10 marks]

A. Our mancala game can be played by repeatedly calling `sow` on a board like so:

```
>>> sow(sow(sow("44444", 2, 3), 3, 2), 0, 4)
'55676'
```

We could also capture the board in a higher-order function which takes in each sowing move, prints the updated board and returns a new higher-order function, like so:

```
>>> make_game("44444")(2, 3)(3, 2)(0, 4)
'44555'
'44566'
'55676'
<function make_game>
```

The inputs of higher-order function are the index and seeds, similar to the function `sow`.

Provide an implementation of the function `make_game`. You may reuse the functions defined in Question 2. [4 marks]

```python
def make_game(board):
    def helper(at, seeds):
        new_board = sow(board, at, seeds)
        print(new_board)
        return make_game(new_board)
    return helper
```

INSTRUCTIONS: Part B should be solved using the given higher-order function, and not by recursion OR iteration.

B. **[Warning: HARD]** It turns out that the function `sow` in Question 2 can be defined in terms of the higher-order function `fold` (given in the Appendix) as follows:

```
def sow(board, at, seeds):
    <PRE>
    return fold(<T1>,
                <T2>,
                <T3>)
```

Please provide possible implementations for the terms T1, T2 and T3. You may optionally define other functions in <PRE> if needed.   [4 marks]

*optional
 <PRE>:

```
def f(n):
    seed = int(board[n])
    if at <= n < at+seeds:
        seed = (seed + 1) % 10
    return str(seed)
```

<T1>:
```
lambda a, b: b + a
```

<T2>:
```
f
```

<T3>:
```
len(board)-1
```

C. Chelsea thinks that the higher-order function `sum` (as given in the Appendix) can be used to define the function `sow` because the `+` operator can be used for both integers and strings.

Explain why her thinking is correct or wrong. [2 marks]

Chelsea is wrong because the sum function given in the appendix has `0` as the base case. While the `+` operator can be used with either string and integer types, it cannot be used with a string and an integer.

Question 4: Trains [23 marks]

INSTRUCTIONS: Please read the entire question clearly before you attempt this problem!! You are also not to use any Python data types which have not yet been taught in class.

A train are made up of railroad cars linked together and pulled by one or more locomotives.

Figure 2: A train.

The basic properties of a railroad car, also known as railcar, to be modeled are its name (which is a string) and its weight in tons (which is an integer).

The following functions support a railcar data type:

- `make_railcar(name, weight)` takes as input the name and the weight of the railcar, and returns a railcar data type.

- `get_name(railcar)` takes as input a railcar, and returns the name of the railcar.

- `get_weight(railcar)` takes as input a railcar, and returns the weight of the railcar.

Example:

```
>>> coach = make_railcar("Passenger Coach", 20)

>>> get_name(coach)
"Passenger Coach"

>>> get_weight(coach)
20
```

A. Explain how you will use <u>tuples</u> to represent a railcar. You may use a box-pointer diagram to aid your explanation. [2 marks]

A tuple with 2 elements, where the first element is the name and the second is the weight.

B. Provide an implementation for the functions `make_railcar`, `get_name` and `get_weight`. [2 marks]

```python
def make_railcar(name, weight):
    return (name, weight)




def get_name(railcar):
        return railcar[0]




def get_weight(railcar):
    return railcar[1]
```

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

A locomotive can be modeled as a railcar with the name `"locomotive"`. The weight property of the locomotive now represents the weight which the locomotive can pull.

The function `make_locomotive(capacity)` takes as input a capacity (which is an integer) and creates a railcar that represents a locomotive that can pull a weight of `capacity` tons. The function `is_locomotive(railcar)` takes as input a a railcar and returns `True` if the railcar is a locomotive and `False` otherwise.

C. Provide an implementation for the functions `make_locomotive` and `is_locomotive`. Remember not to break the abstraction of a railcar. [4 marks]

```python
def make_locomotive(capacity):
    return make_railcar("locomotive", capacity)




def is_locomotive(railcar):
    return get_name(railcar) == "locomotive"
```

[Important!] For the remaining parts of this question, **you should not break the abstraction of a railcar or locomotive in your code.**

D. A train consist of a name (which is a string) and is made up of a sequence of railcars and locomotives. A train data type supports the following functions:

- `make_train` which takes as input the name of the train, and returns a train data type with the given name and no railcars.

- `train_name` takes as input a train and returns the name of the train.

- `add_car` takes as input a train and a railcar and returns a train with the railcar added to the train.

Explain how you will represent a train using either tuples or lists. You may use a box-pointer diagram to aid your explanation. [2 marks]

You can implement a train however you want, as long as it makes sense. Here are two common representations:

1. Use a tuple containing two elements, the first is the name and the second is a tuple of railcars.

2. Use a tuple where the first element is the name and the rest of the elements are the railcars.

E. Provide an implementation for each of the supporting functions of the train data type. [4 marks]

Representation 1:

```python
def make_train(name):
    return (name, ())

def train_name(train):
    return train[0]

def add_car(train, railcar):
    return (train[0], train[1] + (railcar,))
```

Representation 2:

```python
def make_train(name):
    return (name, )

def train_name(train):
    return train[0]

def add_car(train, railcar):
    return train + (railcar,)
```

F. Now a train can only move only if the total capacity of all its locomotives is greater than or equal to the total weight of all its non-locomotive railcars. Recall that the capacity of a locomotive is its "weight" as a railcar.

Write a function `can_move` that takes as input a train, and returns `True` if the train can move, and `False` otherwise.

Example:

```python
>>> midnight_express = make_train("Midnight Express")
>>> coach       = make_railcar("Passenger Coach", 20)
>>> tanker      = make_railcar("Oil Tanker", 50)
>>> locomotive = make_locomotive(100)

>>> midnight_express = add_car(midnight_express, locomotive)
>>> midnight_express = add_car(midnight_express, coach)
>>> midnight_express = add_car(midnight_express, coach)
>>> midnight_express = add_car(midnight_express, coach)
>>> midnight_express = add_car(midnight_express, tanker)
>>> can_move(midnight_express)  # Capacity:100  Weight:110
False
```

```
>>> midnight_express = add_car(midnight_express, locomotive)
>>> can_move(midnight_express)  # Capacity:200  Weight:110
True
```

Provide an implementation of the function `can_move`.   [5 marks]

```
def can_move(train):
    cars = train[1]  # or train[1:] depending on representation
    weight = 0
    for car in cars:
        if is_locomotive(car):
            weight += get_weight(car)
        else:
            weight -= get_weight(car)
    return weight >= 0
```

G. In our implementation we represented a locomotive using a special form of railcar. State one advantage or disadvantage for doing so as opposed to using some another representation for a locomotive. *Hint: Think about the abstraction barriers involved.*   [4 marks]

One advantage is that the elements in our sequence of railcars is homogeneous, that is they are all the same type of object. Since there is no `is_railcar` predicate function for us to check if the element is a railcar or not, we do not have to worry about using the wrong functions on a wrong data type because all our data types are railcar.

One disadvantage is that the special meaning of the `weight` property of a locomotive is not clear. If someone were to treat all the objects as railcars and simply sum their weights, this will give an incorrect value.

Other answers will also be accepted as long as they make sense. This question is open-ended to encourage students to think about the implications of designing data types.

Appendix

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

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


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

Scratch Paper

Scratch Paper

Scratch Paper

— E N D   O F   P A P E R —