SCHOOL OF COMPUTING

ASSESSMENT FOR
Special Term I, 2017/2018

Solutions for CS1010X — PROGRAMMING METHODOLOGY

June 2018 Time Allowed: 2 Hours

INSTRUCTIONS TO STUDENTS

1. Please write your Student Number only. Do not write your name.
2. The assessment paper contains **FIVE (5) questions** and comprises **TWENTY-ONE (21) pages**.
3. Weightage of questions is given in square brackets. The maximum attainable score is 100.
4. This is a **CLOSED** book assessment, but you are allowed to bring **TWO** double-sided A4 sheets of notes for this assessment.
5. Write all your answers in the space provided in this booklet.
6. **Please write your student number below.**

STUDENT NO: _____

(this portion is for the examiner's use only)

Question	Marks	Remark
Q1		
Q2		
Q3		
Q4		
Q5		
Total		

Question 1: Python Expressions [24 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. If the interpreter produces an error message, or enters an infinite loop, explain why.

A. [4 marks]

```
a = [4,2,0,3,4]
b = [a]*5
for i in a:
    if i%2==1:
        b[i] = [4,2,7,3,4]
c = list(b)
print(tuple(map(lambda x: c[x] is a, list(range(len(a))))))
```

`(True, True, True, False, True)`

This is to test that the student understands `map` and list mutation/aliasing.

-2 marks for returning list instead of tuple.

B. [4 marks]

```
a = [1,2,3,4,5]
a[4] = a
b = [a,a,a]
b.extend(b)
b[1][2] = b
print(b[3][2][1][0])
```

`1`

This is to test that the student is able to trace through self-referencing list references.

C. [4 marks]

```
try:
    x = []
    y = 42047
    x.extend(str(y))
    for i in x:
        y = y%int(i)
except:
    print("Got error!",x,y)
else:
    print("Heng ah. Survived.",y,x)
```

```
Got error!  ['4', '2', '0', '4', '7'] 1
```

This is to test that the student understands exception handling syntax and extending a `str` will yield individual characters.

D. [4 marks]

```
x = lambda x: x+2
f = lambda y: x(x(y))
g = lambda y: lambda x: y(y(y(x)))
print(g(f)(2))
```

```
14
```

This is the last chance to allow the student to demonstrate mastery over the lambdas given that many still didn't get it after the midterms and re-midterms.

E. [4 marks]

```
def new_if(pred, then_clause, else_clause):
    if pred:
        then_clause
    else:
        else_clause
def p(x):
    if x>20:
        print(x)
    else:
        new_if(x>5, print(x), p(2*x))
p(4)
```

```
4
8
16
32
```

This is to test that the student understands that `if` is a special form. If we try to define our own `if`, all the arguments will be evaluated before a function call is made.

F. [4 marks]

```
def foo(x,*y):
    d = {x:1,1:2,2:1}
    for i in y:
        d[i+1] = d[d[i]]
    print(sum(d.values()))
foo(0,1,2,3,4)
```

```
9
```

This is to test that the student understands `*args` and also dictionary operations with indirect references. The following is the evolution of the dictionary `d`:

```
i = 1, d = {0:1,1:2,2:1} => d = {0:1,1:2,2:1}
i = 2, d = {0:1,1:2,2:1} => d = {0:1,1:2,2:1,3:2}
i = 3, d = {0:1,1:2,2:1,3:2} => d = {0:1,1:2,2:1,3:2,4:1}
i = 4, d = {0:1,1:2,2:1,3:2,4:1} => d = {0:1,1:2,2:1,3:2,4:1,5:2}
```

Sum of values = $1+2+1+2+1+2=9$.

Question 2: Binary Search Trees Deja Vu [37 marks]

We discussed binary search trees in Lecture 10 and you saw them in Mission 10. In this question, we will check if you fully understood what you should have learnt. First, let's start with the following definitions for the binary search tree:

```
def make_node(entry, left, right):
    return [entry, left, right]

def entry(tree):
    return tree[0]

def get_left(tree):
    return tree[1]

def get_right(tree):
    return tree[2]

def set_left(tree,x):
    tree[1]=x

def set_right(tree,x):
    tree[2]=x

def insert(tree, e):  # Inserts an element e into tree
    # See question 2B
```

A. **[Warm Up]** You may assume that a tree with one element `e` is created with:

```
make_node(e, None, None)
```

Implement the function `make_tree` that takes a list of integers and creates a binary search tree consisting of the integers in the list. [3 marks]

```
def make_tree(lst):
    tree = make_node(lst[0], None, None)
    for e in lst[1:]:
        insert(tree,e)
    return tree
```

-1 mark if wrong solution when `lst` has one element.
-1 mark for using `lst` instead of `lst[1:]` in the loop.

B. Implement the function `insert(tree, e)` that inserts a new element `e` into an existing tree according to the order in the list. Assume that our tree will put the smaller elements on the left. [6 marks]

```
def insert(tree, e):
    if e <= entry(tree):
        left = get_left(tree)
        if left==None:
            set_left(tree,make_node(e,None,None))
        else:
            insert(left, e)
    else:
        right = get_right(tree)
        if right==None:
            set_right(tree,make_node(e,None,None))
        else:
            insert(right, e)
```

-2 mark for not using `make_node` in `set_left/set_right`.
-3 marks for ignoring `set_left/set_right`.

C. Let the number of elements in a tree be n. What is the average and worst case order of growth in time and space for your implementation of `insert` in Part (B)? Explain. [4 marks]

Time (average): $O(\log n)$ for balanced tree.

Time (worst): $O(n)$ for completely unbalanced tree.

Space (average): $O(logn)$ this should be the depth of tree since the solution is recursive

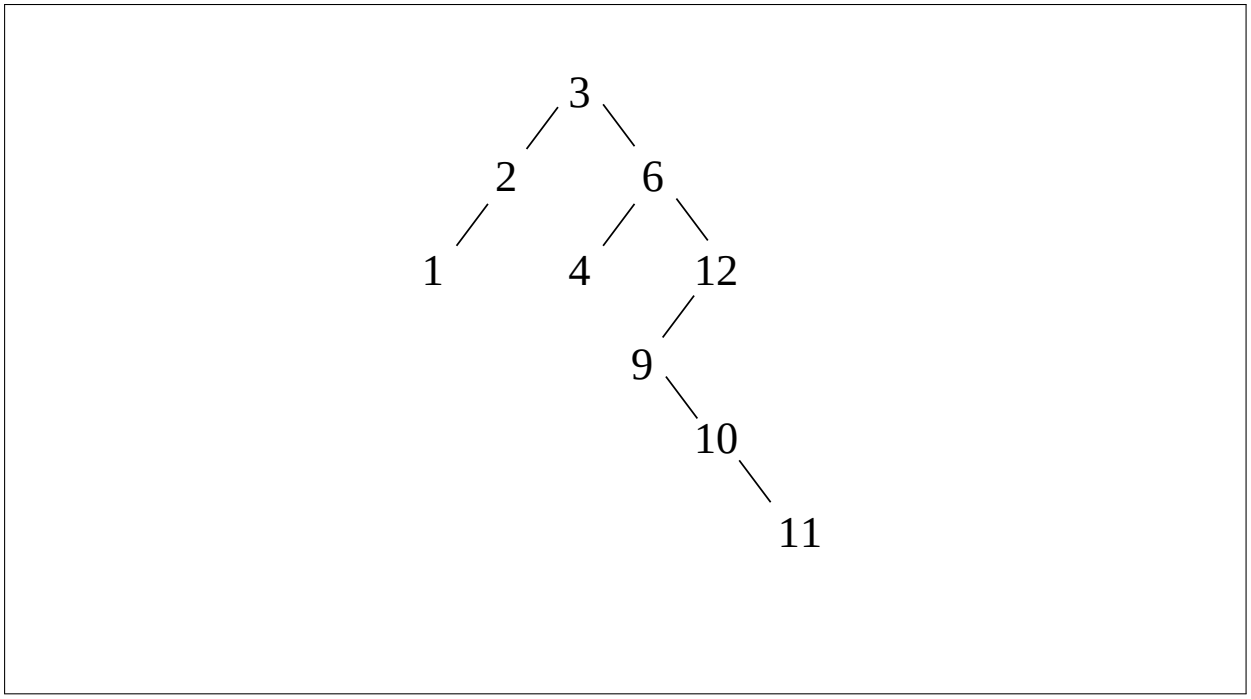
Space (worst): $O(n)$ for completely unbalanced tree

Some students wrote O(1) for both the space complexities. We did not deduct marks for them because the original intention of the question was for students to .

D. Suppose we created the following tree with `make_tree`:

```
t = make_tree([3,2,1,6,12,9,4,10,11])
```

Draw the structure of the resulting tree. [4 marks]



Next, let us define 2 useful helper functions `depth` and `flatten`, that work as follows:

```
>>> t = make_tree([3,2,1,6,12,9,4,10,11])
>>> depth(t)
6
>>> flatten(t)
[1, 2, 3, 4, 6, 9, 10, 11, 12]
```

E. Implement the function `depth` that returns the number of levels in a binary search tree. [3 marks]

```
def depth(tree):
    if tree == None:
        return 0
    else:
        return 1+max(depth(get_left(tree)), depth(get_right(tree)))
```

-2 marks if the student forgot to take the max between left and right. -1 mark if the student forgot to +1.

F. Implement the function `flatten` that returns all the elements in a binary search tree in the correct order. [3 marks]

```
def flatten(tree):
    if tree == None:
        return []
    else:
        return flatten(get_left(tree))+ [entry(tree)] \
        + flatten(get_right(tree))
```

We are testing if the students have mastered tree recursion.
-1 mark if the student use `entry(tree)` instead of `[entry(tree)]`. -1 mark if the base case is wrong.

G. Ben Bitddiddle became very excited after he learnt about binary search trees and `flatten`. He said, we can sort a list by inserting its elements into a BST and then flattening it!! Will the result of sorting using your BST implementation in Parts (A) and (B) be stable? If so, explain. If not, describe how we can make the sort stable. [3 marks]

The current implementation of `insert` in Part (B) will result in an unstable sort. To achieve a stable sort, elements of the same value need to be inserted on the right. This can be achieved by replacing <= with <.

-2 marks if answer yes but did not explain correctly.
-2 marks if answer no but did not give a solution.

H. The tree in Part (D) is unbalanced. Explain what would be a balanced tree and implement the function `balance_tree` that will take a tree and transform it into a balanced tree. [5 marks]

```
>>> t = make_tree([3,2,1,6,12,9,4,10,11])
>>> depth(t)
6
>>> balance_tree(t)
>>> depth(t)
4
```

A balanced tree will have roughly the same number of elements the subtrees of all its nodes.

```
def balance_tree(tree):
    def helper(lst):
        if lst == []:
            return None
        middle = len(lst)//2
        return make_node(lst[middle], helper(lst[:middle]), \
               helper(lst[middle+1:]))

    sorted_list = flatten(tree)
    tmp = helper(sorted_list)
    tree.clear()
    tree.extend(tmp)
```

We are testing if the students have really understood what rebalancing means and whether they can do recursion on trees correctly.
1 mark for the effort of flattening the tree and splitting into halves.
1 mark for the correct explanation.
-1 mark if the student did not say "for each/for all node(s)" in their explanation.

While we expect the students to mutate the tree, we admit that the instructions might not be completely clear, so full marks for returning a new tree.

[**Immutable Funness**] Suppose we now change the implementation of the binary search tree by modifying `make_node` so that it uses a tuple instead of a list:

```
def make_node(entry, left, right):
    return (entry, left, right)
```

Clearly, `entry`, `get_left` and `get_right` still work like before. `set_left` and `set_right` no longer work.

I. Which of the other functions among the following will need to be modified to support this new immutable implementation?

- `make_tree(lst)`
- `insert(tree, e)`
- `depth(tree)`
- `flatten(tree)`
- `balance_tree(tree)`

Provide the updated definitions. [6 marks]

This question verifies if the students understand how immutability affects their code.

2 marks for stating correctly that it is impossible to modify `balance_tree` to work correctly and that we will need to modify the following 2 functions:

```
def make_tree(lst):  # 1 marks
    tree = make_node(lst[0], None, None)
    for e in lst[1:]:
        tree = insert(tree,e)
    return tree

def insert(tree, e): # 3 marks
    if e <= entry(tree):
        left = get_left(tree)
        if left==None:
            return make_node(entry(tree),make_node(e,None,None), \
                   get_right(tree))
        else:
            return make_node(entry(tree),insert(left, e),get_right(tree))
    else:
        right = get_right(tree)
        if right==None:
            return make_node(entry(tree),get_left(tree),\
                   make_node(e,None,None))
        else:
            return make_node(entry(tree),get_left(tree),insert(right, e))
```

If the student returned a new tree for the `balance_tree` function, 2 marks will be awarded for identifying (1 mark) and implementing (1 mark) the `balance_tree` function instead.

Question 3: Undoing Sins of the Past [22 marks]

It is sometimes helpful if we can automatically undo operations in the event where we make mistakes. In this question, you will implement a new list object called an `UndoList` that can automatically undo past mutation operations.

For simplicity, the `UndoList` supports the following 4 methods:

- `append(x)` – appends an element `x` to the end of the list.
- `set(n,x)` – sets element at index n to `x`. Throws `IndexError` if index is out of bounds.
- `undo` – reverses (undo) the last mutation (i.e. either `append` or `set` operation).
- `show` – returns a list containing all the elements.

The following is an example of how the list works:

```
>>> lst = UndoList()
>>> lst.show()
[]

>>> lst.append(2)
>>> lst.append(3)
>>> lst.append(4)
>>> lst.show()
[2, 3, 4]

>>> lst.set(1,99)
>>> lst.show()
[2, 99, 4]

>>> lst.undo()
>>> lst.show()
[2, 3, 4]

>>> lst.undo()
>>> lst.show()
[2, 3]

>>> lst.undo()
>>> lst.show()
[2]

>>> lst.undo()
>>> lst.show()
[]
```

A. Implement the `UndoList` class. [12 marks]

```
class UndoList:
    def __init__(self):  # 2 marks
        self.stuff = []
        self.undo_stack = []

    def show(self):      # 1 mark
        return list(self.stuff)

    def append(self,x):  # 2 marks
        self.stuff.append(x)
        self.undo_stack.append(("pop",))

    def set(self,n,x):   # 2 marks
        self.undo_stack.append(("set",n,self.stuff[n]))
        self.stuff[n] = x

    def undo(self):      # 5 marks
        if not self.undo_stack:
            return
        op = self.undo_stack.pop()
        if op[0] == "pop":
            self.stuff.pop()
        elif op[0] == "set":
            self.stuff[op[1]] = op[2]
```

In this question, we are testing that the students have an understanding of OOP and subclassing. Also, they are tested on the use of a stack to keep track of the undo operations. They might not need to know that it's a stack. They just need to figure out how to keep track of what needs to be undone.

Many students have implemented undo functionality by maintaining a list called history that stores the entire list before any operation is performed. Such a method works only if the copy of existing list is made before appending it in the history. -5 marks for those students who implement such a scheme without copying the list.

[Limited Undo] It might not make sense to have unlimited undo. Suppose we want a new variant of the undo list `LimitedUndoList`, which takes in a parameter n and allows only the last n steps to be undone as follows:

```
>>> lst = LimitedUndoList(3) # limited to 3 undos
>>> lst.show()
[]

>>> lst.append(2)
>>> lst.append(3)
>>> lst.append(4)
>>> lst.append(5)
>>> lst.append(6)
>>> lst.show()
[2,3,4,5,6]

>>> lst.set(2,99)
>>> lst.show()
[2,3,99,5,6]

>>> lst.undo()
>>> lst.show() # 2 undos left
[2,3,4,5,6]

>>> lst.undo()
>>> lst.show() # 1 undos left
[2,3,4,5]

>>> lst.undo()
>>> lst.show() # no undos left
[2,3,4]

>>> lst.undo()
>>> lst.show()
[2,3,4]
```

B. Note that `LimitedUndoList` can be implemented by extending `UndoList` and overriding some methods. What is the minimum set of methods that have to be overridden for your implementation in Part (A)? [2 marks]

We definitely need to override `__init__` to add the undo limit. In addition, we will need to override at least one function and the minimum function will be `undo`. See Part (C) for solution.

Some students have written only `undo` but they do implement both `__init__` in Part (C). They are given full credit.

C. Implement the `LimitedUndoList` class. [8 marks]

The minimal solution is:

```
class LimitedUndoList(UndoList):
    def __init__(self,limit):  # 2 marks
        super().__init__()
        self.limit = limit

    def undo(self):            # 6 marks
        if len(self.undo_stack) > self.limit:
            self.undo_stack = self.undo_stack[-self.limit:]
        super().undo()
```

The following would also be acceptable though marks will be lost in Part (B).

```
class LimitedUndoList(UndoList):
    def __init__(self,limit):   # 2 marks
        super().__init__()
        self.limit = limit

    def append(self,x):         # 3 marks
        self.stuff.append(x)
        self.undo_stack.append(("pop",))
        while len(self.undo_stack)> self.limit:
            self.undo_stack.pop(0)

    def set(self,n,x):          # 3 marks
        self.undo_stack.append(("set",n,self.stuff[n]))
        self.stuff[n] = x
        while len(self.undo_stack)> self.limit:
            self.undo_stack.pop(0)
```

-3 marks if student fail to interpret limited undo function. Limited undo does not mean you can call `undo()` function limited times. It means that undo can only 'undo' limited number of previous operations.
-2 marks if student did not subclass `UndoList` correctly.
-1 mark if students hardcode the limit to 3.

Question 4: Interleaving Funness [14 marks]

A. Given a (Python) list with $2n$ elements, implement a Python function `interleave` that mutates the list so that the first n elements are interleaved with the second n elements. For example:

```
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> interleave(a)
>>> a
[0, 5, 1, 6, 2, 7, 3, 8, 4, 9]

>>> a = list(range(12))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> interleave(a)
>>> a
[0, 6, 1, 7, 2, 8, 3, 9, 4, 10, 5, 11]
```

[4 marks]

```
def interleave(lst):
    left = lst[:len(lst)//2]
    right = lst[len(lst)//2:]
    l,r = 0,0
    for i in range(len(lst)):
        if i%2==0:
            lst[i] = left[l]
            l += 1
        else:
            lst[i] = right[r]
            r += 1
```

-2 marks for returning new list instead of mutating the list.
-1 marks for incorrect bounds.
-1 mark if the code would result in an error, e.g., using / instead of //.
-1 mark for not considering general lists.

[Repeat in C] Now, let's do the same in C. The following is a snippet of C code that attempts to do the same.

```
#include <stdio.h>
#define SIZE 10

void print_array(int a[]) {
    printf("[ ");
    for (int i=0; i<SIZE; i++) {
        if (i!=SIZE-1) {
            printf("%d, ", a[i]);
        } else {
            printf("%d ", a[i]);
        }
    }
    printf("]\n");
}

int main()
{
    int a[] = {1,2,3,4,5,6,7,8,9,10};
    print_array(a);
    interleave(a,SIZE);

    print_array(a);
    return 0;
}
```

The output of this program is:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
[ 1, 6, 2, 7, 3, 8, 4, 9, 5, 10 ]
```

B. The above code is missing the definition for the C function `interleave`. Implement the function `interleave` that achieves the desired output. [6 marks]

```
void interleave(int a[],int size) {
    int left[size/2+1];
    int right[size/2+1];
    int l=0,r=0;
    for (int i=0; i<size; i++) {
        if (i<size/2){
            left[l++] = a[i];
        } else {
            right[r++] = a[i];
        }
    }
    l=r=0;
    for (int i=0; i<size; i++) {
        if (i%2==0){
            a[i] = left[l++];
        } else {
            a[i] = right[r++];
        }
    }
}
```

Basic difference between the Python and C versions is that there is no easy way to copy out the elements in C other than using a loop because there is no slicing.

-2 marks for returning new list instead of mutating the list.
-2 marks if the function signature is wrong.

C. What is the order of growth in time and space for the C function `interleave` you implemented in Part(B) in terms of the number of elements n? Explain. [4 marks]

Time: $O(n)$, since we do a `for` loop over all the elements twice.

Space: $O(n)$, since we need to allocate 2 arrays of size equivalent to the input array.

2 marks for each correct answer.

[Random Musing] As an exercise (for you to think about over the summer): can you come up with an in-place implementation for `interleave` that is faster than $O(n^2)$?

Question 5: CS1010X has Talent [3 marks]

Write us a song (or poem) that articulates what you have learnt in CS1010X over the past 6 months. If you can really cannot sing or hold a tune, you can draw us a picture, but it better be good!

The student will be awarded points as long as he/she is coherent and doesn't say something obviously wrong. Exactly how many points depends on the effort and thoughtfulness put into writing this mini-essay. Amuse the prof and you get full credit.

— E N D O F P A P E R —

Scratch Paper

– H A P P Y H O L I D A Y S ! –