SCHOOL OF COMPUTING

ASSESSMENT FOR
Special Term I, 2021/2022

Solutions for CS1010X — PROGRAMMING METHODOLOGY

June 2022 Time Allowed: 2 Hours

INSTRUCTIONS TO STUDENTS

1. Please write your Student Number only. Do not write your name.
2. The assessment paper contains **8 questions** and comprises **15 pages**.
3. Weightage of questions is given in square brackets. Total score is 48.
4. This is an **OPEN** book assessment, but you are NOT allowed to use any electronic devices such as laptop, mobile phone, and calculator.
5. Write all your answers in the space (enclosed in a box after each part/question) provided in this booklet.
6. You should make effort to write your answer legibly and neatly to avoid any confusion in interpretation. Use pencil where applicable (to present your Python / Java code with proper indentation) to ease any amendment along the way.

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

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

Question	Marks
Q1	
Q2	
Q3	
Q4	
Q5	
Q6	
Q7	
Q8	
Total	

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

If we think of a data structure to be an ingredient and an algorithm to be a recipe, we can relate IDLE (or any other development environments) to be a _____ **kitchen** _____ .

Question 2: Python Expressions [10 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).

A. [4 marks]

```
def expansion_of(a):
    b = []
    c = []
    for item in a:
        b.extend(item)
        c.append(item)
    print (b)
    print (c)
    return

a = ( (1,2,3), ((4,5),), (((6,)),) )
print(expansion_of( a ))
```

```
[1, 2, 3, (4, 5), (6,)]
[(1, 2, 3), ((4, 5),), ((6,),)]
None
```

The "None" is 1 mark, the other 2 are 1.5 each. 0.5 is given to each tuple (there are 3 tuples).

B. [4 marks]

```
def change(a_lst):
    a_lst[1] = "88"
    a_lst = a_lst[2]
    return a_lst

t = ("t1", "t2")
lst = ["a", "b", t]
d = ( ("d1", 1), ("d2", 2), ("d3", lst))
D = dict(d)
t = (1,2)
lst[0]=t
lst[1]=t[1]
d = change(lst)
print(t)
print(lst)
print(d)
print(D)
```

```
(1, 2)
[(1, 2), '88', ('t1', 't2')]
('t1', 't2')
{'d1': 1, 'd2': 2, 'd3': [(1, 2), '88', ('t1', 't2')]}
```

Note: Each output from print is 1 mark.

C. [2 marks]

```
def CV(t1, t2, term):
    result = []
    t = dict(t1)
    for m1, m2, m3 in t2:
        if m1 in t:
            result.append( (m1, t[m1], m2, m3, term(m1, (m1, m2, m3))))
    return result

t1 = ( (1, 10), (2, 20), (2, 21), (6, 60) )
t2 = ( (2, "B", 200), (1, "A", 100), (3, "C", 300),
       (1, "Z", 101), (2, "Y", 201) )

J = CV(t1, t2, lambda x, y: x + y[2])
for tup in J:
    print(tup)
```

```
(2, 21, 'B', 200, 202)
(1, 10, 'A', 100, 101)
(1, 10, 'Z', 101, 102)
(2, 21, 'Y', 201, 203)
```

Note: The tuple (2,21) overrides the tuple (2,20) in the creation of the dictionary t.

Question 3: Iteration & Recursion, with Time & Space [9 marks]

Suppose you are given the following:

```
def seq(a, f, g, c):
    sum = 0
    while a != 0:
        if a > c:
            f, g = g, f
        sum += f(a)
        a = f(a)
    return sum
```

A. [1 marks]

State the output of `seq(10, lambda x: 4*x, lambda x: x//5, 100)`

239

Note: This part is here to help you understand the execution of the function so as to prepare you for the next few parts.

B. [2 marks]

What is a good upper bound of the time complexity of `seq(M, lambda x: 4*x, lambda x: x//5, N)` in terms of some integers `M` and `N` where `0 < M < N`? Your answer should be supported with good justification/reasoning to receive credit.

This is O($\log_4 N$) to get a to reach from M to N, and then O($\log_5 N$) to get the value of a back to 0. So, the time complexity is O($\log_4 N$), which is O($\log N$).

Notice that when $a > c$, the swap of f and g happens, but then no more swapping is needed as a is always less than c after that as before a was multiple by 4 to get larger than c but immediately divided by 5 to become smaller than c, and a stays to be smaller than c from there on.

C. [4 marks]

Provide a recursive implementation of the given function that produces the same output value when called with the same set of inputs.

```
def recur (a, f, g, c):

    if a == 0:
        return 0
    if a > c:
        f, g = g, f
    return f(a) + recur(f(a), f, g, c)
```

D. [2 marks]

State and explain whether your above implementation `recur` has a different time complexity and space complexity from the given function `seq` when the inputs are the same as before: `(M, lambda x: 4*x, lambda x: x//5, N)`. No credit will be awarded if your `recur` is far from correct.

There is no change in the time complexity as the above recursive implementation still repeatedly calculates some same values in each recursive call. But, the space complexity is now in the order of the recursive calls as values of $f(a)$ need to be stack up along the recursive calls. So, the space complexity is now $O(\log_4 N) = O((\log N)$.

Question 4: Matrix Once Again [7 marks]

As promised, we now see yet another representation of a matrix. This representation uses 3 lists `V`, `C`, `R`. First, `V` is the list of non-zero values in the matrix, `C` is the list of column numbers of those corresponding values in `V`. As for which rows are these values, that is the purpose of `R`. Specifically, elements of `V` that are in the i^{th} row are of indices from `R[i]` till `(R[i+1]-1)`. Note that we number the rows and columns starting with 0 instead of 1. Let's consider the below example:

```
V = [ 10, 20, 30, 40, 50, 60, 70, 80 ]
C = [  0,  1,  1,  3,  2,  3,  4,  5 ]
R = [  0,  2,  4,  7,  8]
```

With `R[1]=2`, `R[2]=4`, then `V[2]`, `V[3]` are values from row 1. As `C[2]=1` and `V[2]` is for row 1, so we have the matrix value at [1][1] is `V[2]=30`. And, `C[3]=3` and `V[3]` is for row 1, so we have the matrix value at [1][3] is `V[3]=40`. Below is the Python code to convert this representation into our familiar one using list of lists.

```
def create_matrix(V, C, R):
    numRow = len(R) - 1
    numCol = C[-1] + 1
    row = [0]*numCol
    mat = []
    # get empty matrix
    for i in range(numRow):
        mat.append(list(row))
    # get matrix with values in V
    for i in range(numRow):
        for j in range( R[i], R[i+1]):
            mat[i][C[j]] = V[j]
    return mat
```

A. [1 marks]

Draw out the full (4 rows by 6 columns) matrix of the above given example of `V`, `C`, `R`. The above description should be enough for you to answer this question, but if you need further clarification, then this question is actually asking you to find the output of the function `create_matrix` with the given `V`, `C`, `R`.

```
[ [10, 20,  0,  0,  0,  0],
  [0, 30,  0, 40,  0,  0],
  [0,  0, 50, 60, 70,  0],
  [0,  0,  0,  0,  0, 80]   ]
```

B. [4 marks]

A $n \times m$ matrix M (with n rows and m columns) can multiply with a vector W with m values to get a vector W' with n values. (Just in case you don't know: this matrix-vector multiplication is the same as your familiar matrix-matrix multiplication where W is treated as a $m \times 1$ matrix and W' is the resulting $n \times 1$ matrix.) Now, we take M represented as (`V, C, R`) while W represented as a simple list (i.e a Python list with m values while any zero is also explicitly represented). You are to complete the following function to perform the matrix-vector multiplication to output a vector (which is a list with length equal to n).

```
def M_Vec(V, C, R, W):
    numRow = len(R) - 1
    result = []
    for i in range(numRow):
        sum = 0
        for j in range(R[i], R[i+1]):
            sum += V[j] * W[C[j]]
        result.append(sum)
    return result
```

C. [2 marks]

Please state and explain the time and space complexity of your program in Part B. No credit can be given if your program is far from correct.

For a $n \times m$ matrix with $K = len(V)$ non-zero elements, each element needs to multiply with an element in the vector, for a total of K operations (multiplication and addition). This plus the number of rows n is then the total time complexity of the algorithm, i.e. $O(K+n)$.

As for the space complexity, we need a vector of length n for the output, and it is thus $O(n)$ space.

Question 5: Python – Object Oriented Concepts [6 marks]

We have the following 3 classes:

```
class A(object):
  count = 0
  def __init__(self, name, value):
      self.name = name
      self.value = [value,]
      A.count += 1

  def add_value(self, value):
      for i in range(len(value)):
          self.value[i] += value[i]

  def total_value(self):
      ans = 0
      for i in self.value:
          ans += i
      return ans

class B(A):
  def __init__(self, name, value, asset):
      super().__init__(name, value)
      self.value = self.value + [asset]
      self.count = super().count

  def value_difference(self, other):
      return self.total_value() - other.total_value()

class C(B):
  def __init__(self, name, value, asset, share):
      super().__init__(name, value, asset)
      self.value = self.value + [share]
```

We create 3 objects and perform some calculation on them as follows:

```
a = A("--a--", 10)
b = B("--b--", 20, 10)
c = C("--c--", 30, 20, 10)
b.add_value((5, 2))
c.add_value((5, 2, 1))
```

In each of the following, you are to write the output of the print statement. If there is any resulting error (instead of an output), please state your reasoning for the error.

A. [2 marks]

```
print(b.count, ":", b.total_value())
```

2 : 37

B. [2 marks]

```
print(c.count, ":", c.value_difference(b))
```

3 : 31

C. [2 marks]

```
print(a.count, ":", a.total_value())
```

3 : 10

Question 6: Dynamic Programming Revisit [5 marks]

We now revisit the cut rod problem (in Recitation #10). To recap, you have a rod of length n meters and you hope to cut it into pieces and sell them with maximum profit. You can only cut pieces into integer lengths. The profit of rod with different lengths are as shown in the `prices` variable where, in our example, for a piece of length 1 we can sell it for 1 dollar, length 2 for 5 dollars, length 3 for 8 dollars and so on. The following code is re-produced from that given in our recitation.

```
def cut_rod(n, prices):
    max_price = [0] * (n+1)
    for length in range(1, n+1):
        for p in prices:
            if p <= length:
                max_price[length] = max(max_price[length], prices[p] + \
                                        max_price[length-p])
    return max_price[n]

n = 4
prices = {1: 1, 2: 5, 3: 8, 4: 9, 5: 10, 6: 17, 7: 17, 8: 20, 9: 24, 10: 30}
print( cut_rod(n, prices) )   # output is 10
```

In our question here, we want to work with some restriction on the number of pieces of certain lengths that we can cut our rod. For example, if we are allowed to have only at most 1 piece of length 2, then for a rod with length 4, we cannot cut it into 2 pieces of length 2 to sell for 10 dollars. Instead, we now should cut it into a piece of length 3 and a piece of length 1 to sell them for 9 dollars. So, the result of this restriction is that we cannot cut our rod into any piece of length 2 for a maximum profit.

A. [1 marks]

Now, your turn to work on a different example: Suppose we have a rod of length 6 and with the above `prices`. Our restriction is that we are not allowed to cut it into pieces of length 4 or longer. Also, we can cut it into at most 1 piece of length 3. How should the rod be cut into pieces so that we can sell them for the maximum profit with the given restrictions?

In this case, the best is to cut our rod of length 6 into 3 pieces of length 2 for the maximum profit of 15.

B. [4 marks]

You now want to write a program to solve this type of problem. To make it simple, we just have the restriction on length 3 to at most `num` pieces, and no restriction on other lengths. Please complete the following Python program to use the given `cut_rod` function (possibly more than 1 times if needed). You are not allowed to give any other solution without using the given `cut_rod` function.

```
def cut_rod_limit_3_to(num, n, prices):
    max_price = 0
    p = prices.copy()
    price3 = p[3]
    p[3] = 0    # set it to 0 so that there is no incentive
                # or do a: "del p[3]"
    for i in range(num+1):
        if (n - 3*i) <= 0:
            break # not good enough to use: res = 0
        else:
            res = cut_rod(n - 3*i, p)
        if (res + i*price3) > max_price:
            max_price = res + i*price3
    return max_price

# or

def cut_rod_limit_3_to(num, n, prices):
    max_price = 0
    p = prices.copy()
    price3 = p[3]
    p[3] = 0    # set it to 0 so that there is no incentive
    for i in range(num+1):
        if (n - 3*i) > 0:
            res = cut_rod(n - 3*i, p)
            if (res + i*price3) > max_price:
                max_price = res + i*price3
    return max_price
```

Question 7: Java Exercise [5 marks]

Given the `BankAcct` and `SavingAcct` in our Lecture 15, you are to design a `HousingLoan` class to work with it - you have to decide where this new class of `HousingLoan` is best fit in the hierarchy of classes. (If you do remember the `LoanAcct` class in our lecture training, please ignore its existence - though you may learn something from there - when you answer this question.)

The `HousingLoan` class should be able to capture the following details (and nothing else) with the corresponding methods you have to provide:

1. An integer of an account number
2. A double (negative) number of an outstanding loan amount (which is the full loan amount at the beginning).
3. Number of months remaining for the loan (which is known when the loan is created).
4. Loan interest rate per annum that is known when the loan is created and may be updated along the way due to changes in interest rate.
5. Amount due for payment for the month, which is the outstanding loan amount added with one month of the interest amount, divided by the number of months remaining for the loan. Note that the one month interest amount is equal to the outstanding loan amount multiply by the interest rate per annum divided by 12. After calculated this, we need to adjust the number of months remaining for the loan by one, and add the calculated interest to the outstanding amount (as payment has yet to be done).
6. On some date, the payment due (as calculated in the previous point) will be deducted from a pre-designated bank account (which is given when the loan is created). Once payment is done successfully, the outstanding amount for this housing loan is adjusted accordingly. Note that if a payment is not successful, we simply put up a message (without other penalty computation as in real-life).
7. A lump sum deposit is possible into the loan account to reduce the outstanding loan amount, but no withdrawal is possible with this loan account.

The above is a simplified version of the real housing loan.

Below are the details of the `BankAcct` and `SavingAcct` class taken from our lecture for your reference. Your are to provide the `HousingLoan` class (with skeleton given) in the page after the next.

```
class BankAcct
{
  protected int _acctNum;
  protected double _balance;

  public BankAcct( int aNum, double bal ) {
    _acctNum = aNum;
    _balance = bal;
  }

  public boolean withdraw( double amount ) {
    if ( _balance < amount ) return false;
    _balance -= amount;
    return true;
  }

  public void deposit( double amount ) {
    if ( amount <= 0 ) return;
    _balance += amount;
  }

  public void print() {
    System.out.println( "Account Number: " + _acctNum );
    System.out.printf( "Balance: $%.2f\n", _balance );
  }
}

//Subclass
class SavingAcct extends BankAcct
{
  protected double _rate;
  public SavingAcct(int aNum, double bal, double rate) {
    super( aNum, bal );
    _rate = rate;
  }

  //New method in subclass
  public void payInterest()
  {
    _balance += _balance * _rate;
  }

  //Method Overriding
  public void print( ) {
    super.print();
    System.out.printf( "Interest: %.2f\n", _rate );
  }
}
```

```
class HousingLoan extends BankAcct
{
  protected double _rate;
  protected int _noMonth;
  protected double _paymentDue;
  protected BankAcct _linkedAct;

  public HousingLoan(int aNum, double bal, double rate, int noMonth, BankAcct linkedAct)
  { // assume bal is already a negative number
    super( aNum, bal);
    _rate = rate;
    _noMonth = noMonth;
    _linkedAct = linkedAct;
  }

  public void update_interest_rate_to (double rate) {
    _rate = rate;
  }

  public void cal_mthAmtDue() {
    _balance += _balance*_rate/12;
    _paymentDue = -(_balance / _noMonth); // this has to be positive
    _noMonth = _noMonth - 1;
  }

  public void payMthAmt() {
    if (_linkedAct.withdraw(_paymentDue))
        this.deposit(_paymentDue);
    else
        System.out.println("Money not enough");
  }

  // don't need to do the deposit - inherited

  // don't need to do withdraw - inherited

}
```

Note: there are 4 methods + other stuff, so we allocate roughly 1 mark to each case.

Question 8: Computational Thinking [5 marks]

Now that you have officially completed this course related to computational thinking. As a final assessment, you are to extend what you have learned to other possibilities of computation.

Suppose we have a list (you can imagine the kind of list as in Python). Now the difference is that each element of the list is kept in a core (you can think of a core as a computer), and all the cores are line up in a line as in the order of their elements in the list. To do computation, all cores are synchronised (i.e. there is a global clock that all cores use as reference to start a computational step or a round). At one round (of computation step), besides the arithmetic operations, each core can ask for the value of the list's element stored with at most 1 other core at a fixed distance away from itself, and all cores use the same fixed distance when asking in the same round. The immediate left and right cores of a core are of distance 1 away; the immediate left (right, respectively) of the immediate left (right, respectively) of a core is of distance 2 away, and so on.

Consider this problem: We have a list of N elements stored with the N cores. Some (at least 1) elements have valid values and some with no value. We want each core with no value to get a valid value from its closest neighbor measured by distance. When a core (with no valid value) has 2 closest neighbors with valid values, one on the left and one on the right, we use the left one for the core. At the end, each core has a valid value.

A. [2 marks]

Here is one seemingly "good" algorithm: during each round, each core with no valid value asks its left neighbor (if exists) for a valid value, while each core with a valid value simply do nothing (but will pass its valid value to its neighbor if asked). If a core C (with no valid value) does receive a valid value from its left neighbor, the core C takes that as its valid value; otherwise, it needs to go to the next round to repeat the process to ask from its right neighbor (if exists). Such asking of left and then right neighbor is repeated for each core without a valid value till all cores finally have valid values.

What is the maximum number of rounds (worst case) needed with N cores to complete the computation correctly? Please explain with number of rounds in terms of N and not with big-O notation.

The worst case is when there is 1 core with a valid value, and this one core is at the right end, say position N of the list. So, after 2 rounds, the $(N-1)$th core has a valid value from the Nth core. So, in total, we need $2 \times (N-1)$ rounds for all the cores (especially the leftmost core) to receive the valid value originated from the rightmost core.

B. [2 marks]

Here is one seemingly "better" algorithm: pretty much the same as before, except for varying the cores at different distances to ask for valid values: At round 1, each core without a valid value asks its immediate left neighbor, and round 2, asks its immediate right neighbor. Then, at round 3, each core without a valid value asks the core on the left of distance 2 away, and at round 4, asks the core on the right of distance 2 away. Subsequently, for each two rounds, cores double the distance away in asking for valid values. That is, at round 5, a core without a valid value asks the core on the left at distance 4 away; at round 6, ask on the right at distance 4 away; at round 7, ask on the left at distance 8 away; and round 8, ask on the right at distance 8 away and so on.

What is the maximum number of rounds (worst case) needed with N cores to complete the computation here? Please explain with number of rounds in terms of N and not with big-O notation. (Note that this "better" algorithm does not seem trivial but it actually works.)

The worst case is also when there is 1 core with a valid value, and this one core is at the right end, say position N of the list. So, after 2 rounds, only the $(N-1)$th core has a valid value from the Nth core. But then after round 4, we have 4 cores with valid value (from Nth core), and each 2 rounds double the number of cores with valid value. So, in total, we need $2 \times \lceil \log N \rceil)$ rounds.

C. [1 marks]

With this success in the so-called 1D list of elements in the above, we now consider its variant in a 2D list of elements (list of lists), and a 2D set of cores where each core keeps one element (of the list of lists) as before and can ask from another core "vertically" and "horizontally" for possible value. You are now given the chance to come up with one reaction or question about this extension from 1D to 2D that is relevant to what you have learned in this course.

This is actually the parallel "prefix" in 2D, and the algorithm actually does not work. The alternative of "jump flooding" (google search) actually works most of the time but still has mistakes. So, a good answer here is to question the correctness here - there are examples to show that the algorithm does not calculate the correct "transfer" of values to nearest neighbors.

Another possible (though not preferred) "question" is to ask what could be the number of rounds needed for the "flooding" in the 2D case here too.

— E N D O F P A P E R —