CS1010X — Programming Methodology
National University of Singapore

Re-Practical Examination

Time allowed: 2 hours

Instructions (please read carefully):

1. This is **an open-book exam**. Note that the various methods of sequences (tuple, set, list, dict) that you may need are listed in the first few pages of our **Lecture 8**.
2. You are to do your work without any assistance from another intelligent human being and without any online help outside of coursemology.org – if found otherwise, you will receive **ZERO** for this exam and will be subject to other disciplinary actions.
3. This practical exam consists of **3** "topics" printed in **10** pages (inclusive of this cover page).
4. The maximum score of this quiz is **32 marks** and you will need to answer all questions to achieve the maximum score. You are advised to use your time wisely and not being stuck on any part for too long.
5. Your answers should be submitted on Coursemology.org **BEFORE** the end of the exam. If you have any submissions timestamped with a time after the exam has ended, your submission for that question will not be graded. Remember to **finalize** your submissions before the end of the exam. Also, you are to upload your screen recording to Luminus by 5pm on the day of the exam.
6. You will be allowed to run some public test cases to verify that your submission is correct. Note that you can run the test cases on Coursemology.org up to a **maximum of 15 times**. Note that public test cases provided have explicit input and output stated. On the other hand, **we have also listed our evaluation test cases under the public section but without disclosing their inputs and outputs**. This is done so that you have a better feel on how your codes will perform when we do the marking.
7. You are also provided with some templates to work with - but we would like you to use them at your own discretion. If Coursemology.org fails, you need to rename your file `practical-<matno>.py` where `<matno>` is your matriculation number and submit that file to Luminus.
8. Please behave like a good programmer to make your codes readable with good naming convention for the variables and function names. If you do not do so, we reserve the right to deduct small credit even if your program is correct.

GOOD LUCK!

Topic 1: Straight into 2D [11 marks]

As much as we want, we hope to present challenges of similar level of difficulties as you have seen a week ago though we know this is kind of impossible. Still, we are trying as you will see that we have all the necessary elements from last week's exam....with the comfort of using your own laptop.

Violet continues to be interested in encryption and decryption. Since she knows how to deal with one-dimensional already, let's try a (hypothetical) two-dimensional one right away. There are two cipher texts (strings): one for row (`cipherr`), and one for column (`cipherc`). And the plain text of 'A' to 'Z' and '0' to '9' (total 36 characters) are arranged in a 6×6 table. Example in Python codes look like:

```
cipher_r = '062849'

plain =  [ ['0','1','2','3','4','5'],    # '0' row
           ['6','7','8','9','A','B'],    # '6' row
           ['C','D','E','F','G','H'],    # '2' row
           ['I','J','K','L','M','N'],    # '8' row
           ['O','P','Q','R','S','T'],    # '4' row
           ['U','V','W','X','Y','Z'] ]   # '9' row

cipher_c = 'abcdef'      # 'a' column is '06CIOU'
                         # 'b' column is '17DJPV'
                         # 'c' column is '28EKQW'
                         # 'd' column is '39FLRX'
                         # 'e' column is '4AGMSY'
                         # 'f' column is '5BHNTZ'

# so plain text 'M' is to be encrypted as '8e'
#    plain text 'N' is to be encrypted as '8f'
#
#  'HELLO 1 2 3' is encrypted as '2f2c8d8d4a 0b 0c 0d'
```

Note that `plain`, `cipherr` and `cipherc` may be set to other values with other sizes too. That is, for example, the lengths of `cipherr` and `cipherc` are both 6 in our example here but they each may be longer or shorter too in our test cases. The same is true for `plain` where our example is a 6×6 list but in general it can be of a 2D list of other size.

Question 1. Please write a function `encrypt(message,plain,cipherr,cipherc)` to help Violet encrypt a message. Note that each (single) space still remains a (single) space in the encrypted message, and all input characters are uppercase letters or numbers. Just in case you are still not sure of how the encryption is done with the above example on "HELLO 1 2 3", you should read the paragraph after Question 2 to get further understanding. [3 marks]

Question 2. Please write a function `decrypt(message,plain,cipherr,cipherc)` to help to decrypt a given message from Violet. [4 marks]

From the above questions on 2D, Violet realizes the critical skill needed to extend from 2D to higher dimension is to find out the *index* of the character (in `plain`) that she

wants to encrypt given the `plain` as a list of lists for 2D, or a list of lists of lists in 3D, or a list of lists of lists of lists in 4D, and so on. For example, to encrypt `'M'` in the above 2D case, she has to locate `'M'` as in the 3rd row (index 3) named as the `'8'` row, and in the 4th column (index 4) named as `'e'` column, and thus encrypt `'M'` as `'8e'`. In other words, `plain[3][4] = 'M'` and the index of `'M'` is `'3-4'` in 2D, and then use it to refer to `cipherr` and `ciphere` for the corresponding encrypted 2-characters. So, to work on high dimensional encryption and decryption, Violet must first know how to find out the index for any character in a given `plain` text which is represented in a list (of lists of lists of lists of...).

The above motivates us to find index of item in a general list and the list need not necessarily be a full representation of a plain text. Here are such examples of indices of a given list `lst`:

```
>>> lst = [0,[1,2,3,4],[5,6,[7,[8,9]]]]
>>> for i in range(11):
        print("index of", i, "in", lst, "is:", index(lst, i))

index of 0 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 0
index of 1 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 1-0
index of 2 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 1-1
index of 3 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 1-2
index of 4 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 1-3
index of 5 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 2-0
index of 6 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 2-1
index of 7 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 2-2-0
index of 8 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 2-2-1-0
index of 9 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: 2-2-1-1
index of 10 in [0,[1,2,3,4],[5,6,[7,[8,9]]]] is: None
```

Question 3. Please write a function `index(lst,char)` to find out the `index` (with the hyphenated format as shown in the above examples) of the given `char` in the input list `lst`. Hyphenated format refers to a chain of the index numbers (in order) joining together with '-'. For our example: `lst[2][1]=6`, the output of `index(lst,6)` is then '2-1' (while '1-2' is in the wrong order); and `lst[2][2][1][1]=9`, the output of `index(lst,9)` is '2-2-1-1' (while other permutations such as '1-1-2-2', '2-1-2-1' are in the wrong order).

Note that the input `char` can be any character (for purposes in encryption) and not just integer as shown in our examples. Also, just in case there is some package to do this, you are not allowed to use it for this question as you can't import such a package in our autograder. [4 marks]

Okay, we should have enough of encryption and decryption for now. These exercises seem more than just warming up, especially for Question 3 which need good care in using recursion....

Topic 2: Matrix for Rune Inventor [11 marks]

Remember Violet was managing runes with matrices. As we recall, as long as we can control the grids, you can do everything. This refers to setting up a special purpose matrix to pass on to the `mxn_matrix` routine that we worked on last week. For this topic, you don't really need to know about last week's stuff. Just forget about the rune generation, but focus on generating a needed special (integer) matrix with some nice symmetry.

That is, this topic is to generate a list of lists for each of the 4 questions. You are given the following routine to check your generated matrix:

```
def print_matrix(matrix):
    for i in range(len(matrix)):
        print (matrix[i])
```

Note that you are not allowed to use any Python packages to solve these problems too. That is, you are not allowed to use any import statement.

Question 4. Please write a function `make_rotation_matrix(m,n)` to return a $m \times n$ matrix where a row is the rotation (to the left) by 1 position of its previous row, and the first row is just a running number from 0 to `n-1`. [2 marks]

Below are some examples:

```
>>> print_matrix( make_rotation_matrix(5,5) )
[0, 1, 2, 3, 4]
[1, 2, 3, 4, 0]
[2, 3, 4, 0, 1]
[3, 4, 0, 1, 2]
[4, 0, 1, 2, 3]

>>> print_matrix( make_rotation_matrix(6,5) )
[0, 1, 2, 3, 4]
[1, 2, 3, 4, 0]
[2, 3, 4, 0, 1]
[3, 4, 0, 1, 2]
[4, 0, 1, 2, 3]
[0, 1, 2, 3, 4]
```

Question 5. Please write a function `make_symmetrical_matrix(n)` to return a $n \times n$ matrix where the first row is the running number from 0 to `n-1`, and all subsequent rows are also in some running number order (starting at the diagonal positions) with the property that the transpose of the matrix is also equal to the matrix. [3 marks]

Below are some examples:

```
>>> print_matrix( make_symmetrical_matrix(5) )
[0, 1, 2, 3, 4]
[1, 0, 1, 2, 3]
[2, 1, 0, 1, 2]
[3, 2, 1, 0, 1]
[4, 3, 2, 1, 0]
```

```
>>> print_matrix( make_symmetrical_matrix(6) )
[0, 1, 2, 3, 4, 5]
[1, 0, 1, 2, 3, 4]
[2, 1, 0, 1, 2, 3]
[3, 2, 1, 0, 1, 2]
[4, 3, 2, 1, 0, 1]
[5, 4, 3, 2, 1, 0]
```

Question 6. Please write a function `make_concentric_matrix(m,n)` to return a $m \times n$ matrix where the outermost 'ring' are all zeros, then progressively in increasing number of 1 then 2 in ring form into the 'center' of the matrix. That is, when you connect up the same number next to each other (in the same row, or in two adjacent rows), you should see the 'ring' for that number. [3 marks]

Below are some examples:

```
>>> print_matrix(make_concentric_matrix(3,3))
[0, 0, 0]
[0, 1, 0]
[0, 0, 0]

>>> print_matrix(make_concentric_matrix(4,4))
[0, 0, 0, 0]
[0, 1, 1, 0]
[0, 1, 1, 0]
[0, 0, 0, 0]

>>> print_matrix(make_concentric_matrix(5,5))
[0, 0, 0, 0, 0]
[0, 1, 1, 1, 0]
[0, 1, 2, 1, 0]
[0, 1, 1, 1, 0]
[0, 0, 0, 0, 0]

>>> print_matrix(make_concentric_matrix(5,6))
[0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 1, 0]
[0, 1, 2, 2, 1, 0]
[0, 1, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0]

>>> print_matrix( make_concentric_matrix(5, 10) )
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 1, 2, 2, 2, 2, 2, 2, 1, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Question 7. Please write a function `make_diamond_matrix(m,n)` to return a $m \times n$ matrix that exhibits the shape of a 'diamond' like the one in playing cards. That is, when you connect up the same number next to each other in the same row or two adjacent rows (one below the other, or diagonally to each other), you should see close to a (broken) 'diamond' for that number. [3 marks]

Below are some examples for you to figure out the pattern to write your codes:

```
>>> print_matrix(make_diamond_matrix(7,7))
[0, 1, 2, 3, 2, 1, 0]
[1, 2, 3, 4, 3, 2, 1]
[2, 3, 4, 5, 4, 3, 2]
[3, 4, 5, 6, 5, 4, 3]
[2, 3, 4, 5, 4, 3, 2]
[1, 2, 3, 4, 3, 2, 1]
[0, 1, 2, 3, 2, 1, 0]

>>> print_matrix(make_diamond_matrix(8,8))
[0, 1, 2, 3, 3, 2, 1, 0]
[1, 2, 3, 4, 4, 3, 2, 1]
[2, 3, 4, 5, 5, 4, 3, 2]
[3, 4, 5, 6, 6, 5, 4, 3]
[3, 4, 5, 6, 6, 5, 4, 3]
[2, 3, 4, 5, 5, 4, 3, 2]
[1, 2, 3, 4, 4, 3, 2, 1]
[0, 1, 2, 3, 3, 2, 1, 0]

>>> print_matrix(make_diamond_matrix(9,9))
[0, 1, 2, 3, 4, 3, 2, 1, 0]
[1, 2, 3, 4, 5, 4, 3, 2, 1]
[2, 3, 4, 5, 6, 5, 4, 3, 2]
[3, 4, 5, 6, 7, 6, 5, 4, 3]
[4, 5, 6, 7, 8, 7, 6, 5, 4]
[3, 4, 5, 6, 7, 6, 5, 4, 3]
[2, 3, 4, 5, 6, 5, 4, 3, 2]
[1, 2, 3, 4, 5, 4, 3, 2, 1]
[0, 1, 2, 3, 4, 3, 2, 1, 0]

>>> print_matrix(make_diamond_matrix(7,8))
[0, 1, 2, 3, 3, 2, 1, 0]
[1, 2, 3, 4, 4, 3, 2, 1]
[2, 3, 4, 5, 5, 4, 3, 2]
[3, 4, 5, 6, 6, 5, 4, 3]
[2, 3, 4, 5, 5, 4, 3, 2]
[1, 2, 3, 4, 4, 3, 2, 1]
[0, 1, 2, 3, 3, 2, 1, 0]

>>> print_matrix(make_diamond_matrix(8,7))
[0, 1, 2, 3, 2, 1, 0]
[1, 2, 3, 4, 3, 2, 1]
[2, 3, 4, 5, 4, 3, 2]
[3, 4, 5, 6, 5, 4, 3]
[3, 4, 5, 6, 5, 4, 3]
[2, 3, 4, 5, 4, 3, 2]
[1, 2, 3, 4, 3, 2, 1]
[0, 1, 2, 3, 2, 1, 0]
```

Topic 3: Layering can help? [10 marks]

Violet tried out using dictionary to represent matrix last week, but that turned out to be just a good exercise with no real benefit. Now, since searching is a common operation needed for many other computations, she wonders whether anything could be done efficiently, in particular, just to think about a list of numbers (or keys), and one often wants to find out other associated information related to a number (key). To keep our discussion simple, we ignore the part of the associated information but just focus on the problem of searching given a key.

To be adventurous, Violet is NOT going to use the list data structure provided in Python to keep each item, but use a new creation that comes from the class called `Node` in the below code.

```
class Node(object):
    def __init__(self, num, before, after, top, bottom):
        self.num = num
        self.before = before     # node before the current node
        self.after = after       # node after the current node
        self.top = top           # its copy above the current layer
        self.bottom = bottom     # a copy below the current layer
```

A `list` is then a collection of nodes from the class `Node`, linking up (in a linear fashion) using each other's `before` node and `after` node. Remember: from here on, we will talk about "list" as the new creation here and not the one from Python! Three notices are in order: (1) we will explain the `top` and `bottom` in the class `Node` later on, (2) as mentioned in the earlier paragraph, the node should also have the associated information, but it is ignore for simplicity and we only need the key `num` for this exercise, and (3) as you will see, we do have access to the full list as long as we have access to the first node in the list.

Since the list has to be searched often, we will create the list in a sorted manner with key `num`. This is achieved with the following codes. Notice the codes also demonstrate how one should move from one item (or node) to the next with `before` and `after` node.

```
def insert_into(head, num):
    if head == None: # then this is the very first node in the list
        return Node(num, None, None, None, None)

    previous = None
    current = head
    while current != None:
        if num < current.num: # then insert newNode before the current node
            newNode = Node(num, previous , current, None, None)
            if previous != None:
                previous.after = newNode
            if current.before == None:
                head = newNode
            current.before = newNode
            return head
        previous = current
        current = current.after
    # insert newNode after the last node in the current list
    newNode = Node(num, previous, None, None, None)
    previous.after = newNode
    return head
```

```
>>> lst = None
>>> for i in [4,3,16,14,24,2,5,22,17,9,1,6,11,7,18,12,13]:
        lst = insert_into(lst, i)
```

Question 8. Please write a function `allkeys(node)` to return nodes in *hyphenated format* as shown in the below example (for the list `lst` created with the `for` statement before this question). Hyphenated format (similar to the one in Question 3) refers to a listing of the keys in a chain with one after another connected with '-'. [2 marks]

```
>>> print("Current full list is:", all_keys(lst))
Current full list is: 1-2-3-4-5-6-7-9-11-12-13-14-16-17-18-22-24
```

With the same approach on moving from a node to the next node, you can find whether a key exist in the list.

Question 9. Please write a function `searchlayer(node,num)` to return nodes (in hyphenated format) visited to locate the key `num` as shown in the below examples. Note that we do not explicitly state found or not found in our output, just show the listing of nodes visited for each case. For the case of not found, the last node tested (though not passed through) is also to be listed for clarity. For example, in searching for the node with `num=21`, it is not found and the listing is '1-2-3-4-5-6-7-9-11-12-13-14-16-17-18-22', inclusive of node 22. [3 marks]

```
>>> print("Current full list is:", all_keys(lst))
Current full list is: 1-2-3-4-5-6-7-9-11-12-13-14-16-17-18-22-24

>>> for i in range(1,27,5):
        print( "visited nodes in searching for", i, ":", search_layer(lst,i))

visited nodes in searching for 1 : 1
visited nodes in searching for 6 : 1-2-3-4-5-6
visited nodes in searching for 11 : 1-2-3-4-5-6-7-9-11
visited nodes in searching for 16 : 1-2-3-4-5-6-7-9-11-12-13-14-16
visited nodes in searching for 21 : 1-2-3-4-5-6-7-9-11-12-13-14-16-17-18-22
visited nodes in searching for 26 : 1-2-3-4-5-6-7-9-11-12-13-14-16-17-18-22-24
```

From the above exercise (which is familiar to us as a linear time algorithm), the obvious question is to ask whether one can do better than linear time, just like the case of the logarithmic time binary search. It is of course not exciting to Violet in considering binary search as it is already well-known. So, in the following, she wants to experiment with another idea: how about we build some more layers of lists (one on top of the other), each is of a small sample/subset of the original one?

Continuing with our example of `lst`, we can create another layer called `topLst` with the following Python code. Notice now that the roles of `bottom` and `top`: in the `createtop` routine, a node in `lst` has 40% chance to be included/duplicated in the new layer `topLst`. If a new node is included, the new node is linked to `lst` as the original node's `top` node and itself has the original copy of the node as its `bottom` node.

```
import random
random.seed(1)
def create_top(lst):
    topLst = Node(lst.num, None, None, None, lst)
    previous = topLst
    current = lst.after
    while current != None:
        rand = random.randint(0, 10) # generate random number 0,1,2,3,...,9
        if rand <= 3:
            newNode = Node(current.num, previous, None, None, current)
            previous.after = newNode
            current.top = newNode # linking lst to topLst
            previous = newNode
        current = current.after
    return topLst

>>> topLst = create_top(lst)
>>> print("Current topLst:", all_keys(topLst))
Current topLst: 1-2-4-6-14-16-18
```

You can see in the above that keys in `topLst` also appear in `lst`, while not all keys in `lst` appear in `topLst`. Moreover, you can switch in moving along a list "freely" to the other with the help of `bottom` (to move from `topLst` to `lst` which is always `non-None`), and `top` (to move from `lst` to `topLst` if the value of `top` is `non-None`). That means, for example, to search for key 11, one can search on the `topLst` first till '1-2-4-6' (because the next key 14 is already larger than 11) and then move to `lst` with the `bottom` node of key 6 and continue from there '7-9-11' for visiting a total of 7 nodes: '1-2-4-6-7-9-11' instead of the 9 nodes ('1-2-3-4-5-6-7-9-11') shown earlier with just traversing along `lst`. For clarity in our output, (1) we remove the duplicated nodes (node 6 in our illustration) in the different layers and did not count it twice, and (2) we ignore node (node 14 in our illustration) that we did not actually need to "pass through" but just see its key value.

Now, we don't have to end here, we can continue to build another smaller list based on `topLst`, called it `topMostLst` as follows:

```
>>> topMostLst = create_top(topLst)
>>> print("Current topMostLst:", all_keys(topMostLst))
Current topMostLst: 1-4-16

>>> print("Current topLst:", all_keys(topLst))
Current topLst: 1-2-4-6-14-16-18

>>> print("Current full list is:", all_keys(lst))
Current full list is: 1-2-3-4-5-6-7-9-11-12-13-14-16-17-18-22-24
```

Note that the above 3 listings may be different when you run your code as it depends on the random number generated at your machine. So, the content of the list is for our illustration purposes only, and don't worry about them too much as your code and our evaluation cases are to be run on the same machine eventually. Now, with the 3 lists, one on top of the other, if we want to find key 11 starting with the `topMostLst`, the listing of nodes we get should be: '1-4-6-7-9-11', yet another improvement to when we have just

`topLst` and `lst`. Though this may be seen like a very small improvement, it can be very significant when the list is large.

Question 10. Please write a function `search(topMostLst,num)` to return nodes (in hyphenated format) visited to locate the key `num` as shown in the below examples. Note that you could imagine this is like water flowing along a stairway, moving from one layer (`topMostLst`) to the next layer (`topLst`) and then next next layer, till the `num` is found or confirmed its non-existence. We do not explicitly state found or not found in our output, just show the listing of nodes visited for each case. For the case of not found, the last node tested (though not really passed through) is also to be listed for clarity. For example, in search for node with `num=10`, it is not found and the listing is '1-4-5-6-7-9-11', inclusive of the last tested node 11. [5 marks]

```
>>> for i in range(26):
        print("searching", i, ":", search(topMostLst,i) )

searching 0 : 1
searching 1 : 1
searching 2 : 1-2
searching 3 : 1-2-3
searching 4 : 1-4
searching 5 : 1-4-5
searching 6 : 1-4-6
searching 7 : 1-4-6-7
searching 8 : 1-4-6-7-9
searching 9 : 1-4-6-7-9
searching 10 : 1-4-6-7-9-11
searching 11 : 1-4-6-7-9-11
searching 12 : 1-4-6-7-9-11-12
searching 13 : 1-4-6-7-9-11-12-13
searching 14 : 1-4-6-14
searching 15 : 1-4-6-14-16
searching 16 : 1-4-16
searching 17 : 1-4-16-17
searching 18 : 1-4-16-18
searching 19 : 1-4-16-18-22
searching 20 : 1-4-16-18-22
searching 21 : 1-4-16-18-22
searching 22 : 1-4-16-18-22
searching 23 : 1-4-16-18-22-24
searching 24 : 1-4-16-18-22-24
searching 25 : 1-4-16-18-22-24
```

Violet knows very well that the idea need not stop at 3 layers. It can continue as long as there are still many nodes in the current topmost layer. But for now, we should stop here though we can't promise you that we don't test your `search(topTopMostLst,num)` with `topTopMostLst` which is yet another layer above `topMostLst` etc.

In any case, by now, we should all be exhausted by Violet for keeping us busy with 1D, 2D, and high dimensions.

Are we going to deal with such issue in the coming days? Most likely, as 1D is really easy to handle by you for sure...

— E N D O F P A P E R —