SCHOOL OF COMPUTING

Take-Home Assignment For
Special Term I, 2019/2020

Solutions for CS1010X — PROGRAMMING METHODOLOGY

June 2020 Time Allowed: 2 Hours

INSTRUCTIONS TO STUDENTS

1. You are to do your work without any assistance from another intelligent human being, and to submit your work yourself with your coursemology account before the time is out – if found otherwise, you will receive zero credit for the whole assessment and subject to other disciplinary action to be taken against you.
2. The assessment paper contains **FOUR (4) questions** and comprises **FOURTEEN (14) pages**.
3. Weightage of each question is given in square bracket. The maximum attainable score is 30, with a bonus question of 3 marks at the end of the paper.
4. You should plan and type your answer neatly to avoid any possible confusion in interpretation. Do format your Python code and provide comments to your codes where applicable. You should not be using the provided code templates for debugging purposes as this is not a Practical Exam; otherwise, you would run the risk of not having enough time to complete the assessment. The templates are provided for easy reference to format your code, and simple checking only.
5. For expressing, for example, time and space complexity, you may use "_" to mean subscript, and " ˆ " to mean superscript. For example, you can write "log_2 *m*" to mean $\log_2 m$, and "2ˆ3" to mean 2^3.
6. For any question that you cannot solve fully, partial credit can still be awarded for any reasonable thoughts you have put down in attempting the question.
7. This is an OPEN book assessment and internet search is allowed. No credit can be given if you cut-and-paste passages or codes from the web regardless of their relevance to the question. You are to present your understanding for all cases. Also note that the paper is set without any provision on time needed by you to do any internet search; that is, you should do away with any temptation to search the internet for answers to the questions.
8. **Do save your work as draft now and then where applicable in coursemology, and finalize it only at the end. Also, you should keep your answer on your computer too in case of need to cut-and-paste into coursemology again.**

Question 1: Iteration and Recursion [10 marks]

Suppose you are given the following function:

```python
def fun(n, m):
    if n==0:
        return 0
    elif n%m==0:
        return n + fun((n-1)//m, m)
    else:
        return n + fun(n-1, m)
```

A. What is the time complexity of the function `fun` in terms of n and m? You should provide the time complexity for all ranges of m: when $m = 1$; when $1 < m < n$; and when $m \geq n$. Justify your answer for each case. You can assume each "%" and "//" operation costs $O(1)$ time. [3 marks]

Time complexity when $m = 1$: $O(n)$.
Time complexity when $1 < m < n$: $O(m \log_m n)$.
Time complexity when $m \geq n$: $O(n)$. (When $m = n$, it is O(1) to be precise.)

Justification (for each case):
When $m = 1$, the `elif` branch of the condition is executed always, and it is of the same behavior as the `else` branch to reduce n by 1 each time. So, it needs $O(n)$ time.

When $m > n$, the `else` branch is executed and it reduces n by 1 each time. So, it needs also $O(n)$ time. When $m = n$, the `elif` is executed with $n = 0$ passed into the function, so O(1) which is also O(n).

When $1 < m < n$, we have the following argument. The `else` branch of the condition is executed for at most $(m-1)$ time then it must reach the `elif` branch. The `elif` branch is to keep only a fraction of n to continue (you are familiar with $m = 2$ that is halving the n). So, there are only at most $\log_m n$ times possible to take an m-fraction till n is 0, and each time we do at most $(m-1)$ reduction of n at the `else` branch to reach another time of the `elif` branch. So the time is $m \log_m n$.

B. Provide an <u>iterative</u> implementation of `fun` to achieve the same output as in the previous part. [2 marks]

```python
def fun_itr(n, m):

    result = 0
    while (n > 0):
        result += n
        if n%m==0:
            n = (n-1)//m
        else:
            n = n-1
    return result
```

Now, suppose you are given the following functions:

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)

def fun2(n):
    result = 0
    for i in range(1, n):
        fact_i = fact(i)
        for j in range(1, i):
            result += fact_i + fact(j)
    return result
```

C. Provide the time complexity of the function `fun2` in terms of n. Justify your answer. [3 marks]

Time complexity: $O(n^3)$

Justification:
The calling of `fact(k)` happens $(n-k)$ times for k from 1 to $n-1$. That is, `fact(1)` is done just $(n-1)$ times (when $i=1$, and when $j=1$ for each of $i=2,\dots,n-1$), `fact(2)` is done $(n-2)$ times (when $i=2$, and when $j=2$ for each of $i=3,\dots,n-1$), etc. till `fact(n-1)` is done just once when $i=n-1$. So, the justification for the time complexity is: $1(n-1)+2(n-2)+3(n-3)+\dots(n-1)1$, and we know (from the web) that this sum is $(n-1)\times n\times(n+1)/6$.

See https://www.youtube.com/watch?v=dUs5-Ak4WSw if you are interested in the derivation. Alternatively, you can do an approximation to see that the sum is no larger than $(n-1)\times(n-1)^2$ because each of the $(n-1)$ term is smaller than $(n-1)^2$. So, the sum is bounded above by $O(n^3)$. What about bounded below? We can take those terms starting from first quarter to the middle. Each one is $\geq n/4\times(n-n/2)=n/4\times n/2$, and we have $n/4$ of them here. So, the sum is bounded below by $O(n^3)$ (as we ignore the constant $1/32$).

D. Write a function `fun2_better` that improves the time complexity you calculated in Part C of this question. (**Hint:** Can you avoid calling the function `fact`?) [2 marks]

```python
def fun2_better(n):
    result = 0
    fact_i = 1
    factor = n-1
    for i in range(1, n):
        fact_i = fact_i * i
        result += fact_i * (i-1)
        factor -= 1
        result += fact_i * factor
    return result
```

Many of you use (non-linear time and space) memo or DP approaches that we have to accept reluctantly :)

Question 2: Higher Order Functions [6 marks]

In this question, you will be working with the following higher-order function:

```
def new_fold(op, f, t, next, condition):
    if condition(t) < 1:
        return 0
    else:
        return op( f(t), new_fold(op, f, next(t), next, condition) )
```

A. The function `cross_sum` takes two input tuples τ_1 and τ_2 (of numbers and of the same length) to compute the sum of their corresponding product terms:

$$\texttt{cross\_sum}(\tau_1, \tau_2) = \tau_1(0) * \tau_2(0) + \tau_1(1) * \tau_2(1) + \ldots + \tau_1(n-1) * \tau_2(n-1)$$

where n is the length of τ_1 and τ_2.

Example execution:

```
>>> cross_sum( (1,2,3,4,5), (1,2,3,4,5) )   # 1*1+2*2+3*3+4*4+5*5 =
55

>>> cross_sum( (1,2,3,4,5,6), (1,2,3,4,5,6) )   # 1*1+2*2+3*3+4*4+5*5+6*6 =
91
```

We can define `cross_sum` with `new_fold` as follows:

```
def cross_sum( t1, t2 ):
    return new_fold ( lambda x,y: x+y, <T1>, <T2>, <T3>, <T4> )
```

You are to provide the completed `cross_sum` function by replacing <T1>, <T2>, <T3>, and <T4> with the correct expressions. 1 mark each is allocated to <T1> and <T3>, and 0.5 mark each to <T2> and <T4>. [3 marks]

```
def cross_sum( t1, t2 ):
    return new_fold ( lambda x,y: x+y,
        lambda t: t[0][0]*t[1][0],       # T1
        (t1, t2),                        # T2
        lambda t: (t[0][1:], t[1][1:]),  # T3
        lambda t: len(t[0]) )            # T4
```

Alternative from you (and there are many others):

```
def cross_sum( t1, t2 ):
    return new_fold (lambda x,y: x+y,
                     lambda x: t1[x]*t2[x],
                     len(t1)-1,
                     lambda x:x-1,
                     lambda x: x+1)
```

B. The function `cross_sum_back` takes two input tuples τ_1 and τ_2 (of numbers and of the same length) to compute the sum of products of their terms as follows:

$$\texttt{cross\_sum\_back}(\tau_1, \tau_2) = \tau_1(0) * \tau_2(n-1) + \tau_1(1) * \tau_2(n-2) + \ldots + \tau_1(n-1) * \tau_2(0)$$

where n is the length of τ_1 and τ_2.

Example execution:

```
>>> cross_sum_back( (1,2,3,4,5), (6,7,8,9,0) )  # 1*0+2*9+3*8+4*7+5*6 =
100

>>> cross_sum_back( (1,2,3,4,5,6), (7,8,9,0,1,2) )  # 1*2+2*1+3*0+4*9+5*8+6*7 =
122
```

We can define `cross_sum_back` with `new_fold` as follows:

```
def cross_sum_back( t1, t2 ):
    return new_fold ( lambda x,y: x+y, <T5>, <T6>, <T7>, , <T8>)
```

You are to provide the completed `cross_sum_back` function as a working program by replacing <T5>, <T6>, <T7>, and <T8> with the correct expressions. 1 mark each is allocated to <T5> and <T7>, and 0.5 mark each to <T6> and <T8>. [3 marks]

```
def cross_sum_back( t1, t2 ):
    return new_fold ( lambda x,y: x+y,
        lambda t: t[0][0]*t[1][-1],      # T5
        (t1, t2),                        # T6
        lambda t: (t[0][1:], t[1][:-1]),# T7
        lambda t: len(t[0]) )            # T8
```

Alternative from you (and there are many others):

```
def cross_sum_back( t1, t2 ):
    return new_fold (lambda x,y: x+y,
                     lambda x : t1[(len(t1)-1)-x] * t2[x],
                     len(t1) -1,
                     lambda x : x-1,
                     lambda x : x+1)
```

Question 3: COVID-19 ver 0.2 [10 marks]

Didn't we say in the midterm that we were working on version 0.1? Now lets extend it to version 0.2, where in addition to tracking cases, we also want to be able to identify clusters of patients that have been in close contact. Before we dive into version 0.2, let's have a quick recap of version 0.1, where we built database that holds people's records. Each record in this database of people (henceforth refered to as `peopleDB`) holds the following information:

1. `name` : name of the person (assume this is unique for each person)
2. `home` : place of residence (assume at most one place for each person)
3. `workplace` : place of work (assume at most one place for each person)
4. `caseType` : Normal (`"n"`), Quarantined (`"q"`) or Confirmed (`"c"`)
5. `places` : places visited by the person - it can be zero or many locations.

We also defined a number of functions to manipulate the entries in this database. You can feel free to reuse these functions for this version 0.2. **If you are already familiar with version 0.1 in the midterm, you can skip this page to continue in the next for the requirements on version 0.2.**

1. `make_empty_db` : This function takes no arguments and returns an empty database of people.
2. `add_person` : This function takes a `peopleDB` and a person's `name`, `home`, `workplace`, and `caseType` and returns a new database with the entry of the new person inserted.
3. `remove_person` : This function takes in a `peopleDB` and a person's `name`. If the person with the given name exists, the function returns a new database with the entry for the specified person removed.
4. `same_home_or_office_as` : This function takes in a `peopleDB` and a person's `name`. It returns a tuple of `names` of people (excluding the said person) staying in the same `home` location or working at the same `workplace`. It can happen that a `home` location for one person is a `workplace` for another and vice-versa. In such cases, these people will be included in the returned tuples of names.
5. `add_visited_places` : This function takes in a `peopleDB`, a person's `name` and a tuple of `places` visited by this person. These places are recorded in the said person's record and a new database with this updated is returned.
6. `same_visited_places_as` : This function takes in a `peopleDB` and a person's `name` to return a tuple of `name` of people (excluding the said person) who have visited at least one same place as the given person.
7. `set_case_to_quarantined` : This function takes in a `peopleDB` and a person's `name` to set the `caseType` of this person to `"q"` if this person is not already a confirmed case. It returns the database with this record updated.

8. `set_case_to_confirmed` : This function takes in a `peopleDB` and a person's `name` to set the `caseType` of this person to `"c"`. In addition, this function also sets the `caseType` of all people who have either visited the same places or work or stay in the same locations as the specified person to `"q"` (Quarantined) if they are not already confirmed cases. It returns the database with these records modified.

For version 0.2, we will be creating a supporting database called `clusterDB` to track different clusters of patients. A cluster is simply defined as a group of people who have visited a common `places` or have a common `workplace` or `home` location with at least one other person in that cluster. Note that since people visited multiple places and work and stay in different locations, they can exist in multiple clusters (This simple definition is to make your life easier – In the real world, the cluster definition is likely to be different.)

You need to write the following functions:

1. `make_empty_clusterDB` : This function takes no input to return a cluster database with the necessary initialization depending on your choice of data structure.
2. `get_number_of_cases` : This function takes a `clusterDB` as input to return the total cases recorded in the cluster database.
3. `get_number_of_clusters` : This function takes a `clusterDB` as input to return the number of clusters in the cluster database.
4. `get_people_in_largest_cluster` : This function takes a `clusterDB` as input and returns a tuple of `name` of people who are part of the largest cluster. Ties between multiple clusters with the largest number of people can be broken arbitrarily (your function can choose anyone of these as the largest cluster).
5. `add_person_to_clusterDB` : This function takes a `peopleDB`, a `clusterDB` and a person's `name`. It should return a new `clusterDB` with the said person added to the appropriate clusters. Adding a person to clusters comprises of the following:

   (a) **Assigning a case number:**
   Every new case added to the `clusterDB` must be assigned a `case_number` that is one larger than the case number of the last person added to `clusterDB`. Case numbers start with 1.

   (b) **Determine links and add to clusters:**
   Next you must determine which clusters the new case should be added to. Going through each person in a cluster, you identify if the new person shares a `workplace` or `home` or has the same visited `places` as anyone in the cluster. If so, the new person should be added to the said cluster; if no, move on to check people in other clusters.

   The `case_number` of this anyone that caused this new person to be in the same cluster is stored in `linked_to` with the new person in the cluster. Figure 1 in the next page should make this process clearer.

   In case of multiple links in a single cluster, you can choose arbitrarily. In case of no links to anyone in any existing clusters, a new cluster should be created for this new person, with `linked_to` set to `0`.

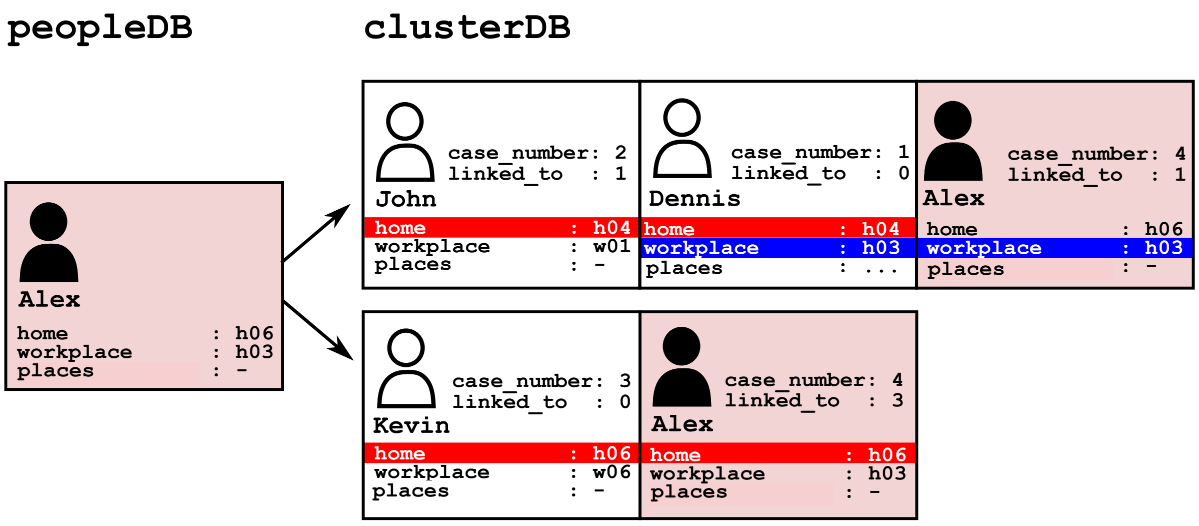

Figure 1: This figure illustrates how we define cluster membership in our `clusterDB`. `"Alex"` shares his workplace with `"Dennis"` in the first cluster, and his home with `"Kevin"` in the second cluster. Therefore, the `clusterDB` should record `"Alex"` as a member of both these clusters. You are not required to store `home`, `workplace`, and `places` of each record in the `clusterDB`. These details have been shown here for illustration purposes only.

Important: You are advised to read through all the requirements below before deciding on the implementation of your `clusterDB` database. Continuing from the midterm, we are doing this question in a non-object oriented fashion as given in the following Sample execution.

Sample execution:

```
# make an empty database, and then add Alice, Ben, Cathy, Dennis, John, Kevin
>>> db = make_empty_db()

# the parameters are: db, name, home, workplace, caseType
>>> db = add_person( db, "Alice", "H01", "W01", "n")
>>> db = add_person( db, "Ben", "H01", "W02", "n")
>>> db = add_person( db, "Cathy", "H03", "W01", "n")
>>> db = add_person( db, "Dennis", "H04", "H03", "n")
>>> db = add_person( db, "John", "H04", "W01", "n" )
>>> db = add_person( db, "Kevin", "H06", "W06", "n" )

>>> print( same_home_or_office_as( db, "Alice" ) )
('Ben', 'Cathy', 'John')

>>> print( same_home_or_office_as( db, "Dennis" ) )
('Cathy', 'John')

>>> db = add_visited_places( db, "Dennis", ("VivoCity", "SAFRA", "Jurong East") )
>>> db = add_visited_places( db, "Ben", ("SAFRA", "NUS") )
# both Ben and Dennis have visited SAFRA
>>> print( same_visited_places_as(db, "Dennis") )
('Ben',)
```

```
# Starting from here, we keep confirmed cases into respective clusters
clusterDB = make_empty_clusterDB()

# Very first confirmed case: Dennis
>>> db = set_case_to_confirm( db, "Dennis")
Done confirm: Dennis
Done quarantine: Cathy
Done quarantine: John
Done quarantine: Ben

# Adding Dennis into the cluster database
>>> clusterDB = add_person_to_clusterDB(db, clusterDB, "Dennis")
>>> print("Number of confirmed cases:", get_number_of_cases(clusterDB))
Number of confirmed cases: 1
>>> print("Number of clusters:", get_number_of_clusters(clusterDB))
Number of clusters: 1

# Second confirmed case: John, in the same cluster as Dennis
>>> db = set_case_to_confirm( db, "John")
Done confirm: John
Done quarantine: Alice
Already confirmed before: Dennis -- no quarantined needed
Already quarantined before: Cathy
>>> clusterDB = add_person_to_clusterDB(db, clusterDB, "John")
>>> print("Number of confirmed cases:", get_number_of_cases(clusterDB))
Number of confirmed cases: 2
>>> print("Number of clusters:", get_number_of_clusters(clusterDB))
Number of clusters: 1

# Third confirmed case: Kevin, forming a new cluster
>>> db = set_case_to_confirm( db, "Kevin")
Done confirm: Kevin
>>> clusterDB = add_person_to_clusterDB(db, clusterDB, "Kevin")
>>> print("Number of confirmed cases:", get_number_of_cases(clusterDB))
Number of confirmed cases: 3
>>> print("Number of clusters:", get_number_of_clusters(clusterDB))
Number of clusters: 2

# Find the people in the largest cluster. If there are more than one such
# cluster, just take anyone of them to list out the people in the cluster
>>> print( get_people_in_largest_cluster(clusterDB))
('Dennis', 'John')
```

A. Explain how you are going to keep track of people in each cluster in a cluster database. To help us understand your ADT structure better, illustrate the contents of your cluster database after the insertion of some dummy confirmed cases into the database. [2 marks]

We are going to use a tuple structure to record the current number of cases as its first element, number of clusters as its second element, and then starting from the third element and so on, we keep each cluster's details. Each cluster is also a tuple of tuple where each tuple is a case number, the name of the person, and which case it is linked to. For a case forming a new cluster, the case it is linked to is set to 0. For example, at the end of the sample execution, we have the following tuple in our `clusterDB`:

```
(3, 2, ((1, 'Dennis', 0), (2, 'John', 1)), ((3, 'Kevin', 0),))
```

[-1 mark for no sample insertion]

B. Do the following 4 parts in the same answer box in coursemology. All should be in accordance to what you declare in Part (A) to implement your cluster database.

(i) Write the function `make_empty_clusterDB`. [1 marks]

(ii) Write the function `get_number_of_cases`. [1 marks]

(iii) Write the function `get_number_of_clusters`. [1 marks]

(iv) Write the function `get_people_in_largest_cluster` to output a tuple of `name` of people in the largest cluster. If there are more than one such clusters, you can output people in anyone of them. [2 marks]

```
def make_empty_clusterDB():
    return (0, 0)

# A lot of students simply added up the cases in all the clusters.
#This will not work because a case can belong to multiple clusters.
def get_number_of_cases(clusterDB):
    return clusterDB[0]

def get_number_of_clusters(clusterDB):
    return clusterDB[1]

def get_people_in_largest_cluster(clusterDB):
    maxC = 0
    for i in range(clusterDB[1]):
        if maxC < len(clusterDB[i+2]):
            maxC = len(clusterDB[i+2])
            maxC_number = i
    return tuple(map(lambda x: x[1], clusterDB[maxC_number + 2]))
```

C. Write the function `add_person_to_clusterDB`. This function is to take in a `name` who is a confirmed case to add to the respective cluster. For this part, you should use `same_home_or_office_as(db, name)` and `same_visited_places_as(db, name)` appeared in midterm test. (You should use them as described at the beginning of the question without the need to know their implementation details.) You may want to write a few sentences to comment your logic in solving the problem at the beginning. Do format your code properly and provide meaningful comments where applicable.

[3 marks]

```python
def add_person_to_clusterDB(db, clusterDB, name):
    case_number = clusterDB[0]+1
    same_ho = same_home_or_office_as(db, name) # 1.
    same_pv = same_visited_places_as(db, name) # 2.
    found = False
    for c_number in range(clusterDB[1]): # going through each cluster
        cur_cluster = clusterDB[2 + c_number]
        for person in cur_cluster: # person is (case#, name, linkTo)
            if person[1] in same_ho or person[1] in same_pv:
                # set found so that there is no need to create a new cluster
                found = True
                # add the person into the cluster
                cur_cluster = cur_cluster + ((case_number, name, person[0]),)
                # update the  clusterDB, and move on to check next cluster
                clusterDB = clusterDB[0:2+c_number] + (cur_cluster,) + \
                            clusterDB[2+c_number+1:]
                break

    if not found: # need to create a new cluster
        clusterDB = (clusterDB[0]+1,) + (clusterDB[1]+1,) + clusterDB[2:] \
                    + (( (case_number, name, 0),),)
    else:
        clusterDB = (clusterDB[0]+1,) + clusterDB[1:]

    return clusterDB
```

Question 4: Changing Coins Again [4 marks]

Below is the dynamic programming code for forming some amount with given types of coins (solved in Recitation #10).

```
def dp_cc(a, d):
    table = []
    oneline = [0]*(d+1)
    for i in range(a+1):
        table.append(list(oneline))
    for i in range(1,d+1):
        table[0][i] = 1
    for col in range(1, d+1):
        for row in range(1, a+1):
            if (row - coins[col-1]) < 0:
                table[row][col] = table[row][col-1]
            else:
                table[row][col] = table[row][col-1] + table[row-coins[col-1]][col]
    return table

def print_table(table):
    for i in range(len(table)):
        print(i, ":", table[i])

>>> coins = [1, 5, 10, 20, 50]
>>> magic_table = dp_cc(100, 5)
>>> print_table( magic_table )
```

The output of the last statement on the content of `magic_table` is as follows (where each row is the amount to form, and each column is the types of coins available, for example, column 3 (starting with 0) means [1, 5, 10] cents coins are available. Take for example, we have 84 ways to form 60 cents with unlimited number of 1, 5, 10, and 20 cents coins, and 88 ways if we are also given unlimited 50 cents coins):

```
0  : [0, 1, 1, 1, 1, 1]
1  : [0, 1, 1, 1, 1, 1]
2  : [0, 1, 1, 1, 1, 1]
3  : [0, 1, 1, 1, 1, 1]
4  : [0, 1, 1, 1, 1, 1]
5  : [0, 1, 2, 2, 2, 2]
6  : [0, 1, 2, 2, 2, 2]
7  : [0, 1, 2, 2, 2, 2]
8  : [0, 1, 2, 2, 2, 2]
9  : [0, 1, 2, 2, 2, 2]
10 : [0, 1, 3, 4, 4, 4]
11 : [0, 1, 3, 4, 4, 4]
12 : [0, 1, 3, 4, 4, 4]
13 : [0, 1, 3, 4, 4, 4]
14 : [0, 1, 3, 4, 4, 4]
15 : [0, 1, 4, 6, 6, 6]
16 : [0, 1, 4, 6, 6, 6]
17 : [0, 1, 4, 6, 6, 6]
```

```
18 : [0, 1, 4, 6, 6, 6]
19 : [0, 1, 4, 6, 6, 6]
20 : [0, 1, 5, 9, 10, 10]
...
...
30 : [0, 1, 7, 16, 20, 20]
...
...
40 : [0, 1, 9, 25, 35, 35]
...
...
50 : [0, 1, 11, 36, 56, 57]
...
...
60 : [0, 1, 13, 49, 84, 88]
...
...
70 : [0, 1, 15, 64, 120, 130]
...
...
80 : [0, 1, 17, 81, 165, 185]
...
...
90 : [0, 1, 19, 100, 220, 255]
91 : [0, 1, 19, 100, 220, 255]
92 : [0, 1, 19, 100, 220, 255]
93 : [0, 1, 19, 100, 220, 255]
94 : [0, 1, 19, 100, 220, 255]
95 : [0, 1, 20, 110, 250, 294]
96 : [0, 1, 20, 110, 250, 294]
97 : [0, 1, 20, 110, 250, 294]
98 : [0, 1, 20, 110, 250, 294]
99 : [0, 1, 20, 110, 250, 294]
100: [0, 1, 21, 121, 286, 343]
```

A. The above assumes there is no limit to the number of each type of coin you can use. Now, suppose we have only one 5 cents coin, and unlimited numbers of 1 cent, 10 cents, 20 cents, and 50 cents coins. How many ways are there now to form 11 cents? List out all the ways one by one. (Hint: Easy as there are not many – just do it manually without referring to the above listing.) [1 marks]

B. Why the above is called a "magic" table? This is because the problem with limited coins can actually be solved directly through numbers in the `magic_table`. How? You should find this out for your answer to Part (A) (and try a few more examples of small amounts, say less than 20 cents, etc. to help in your understanding). Now, with input `num` on the limited number of 5 cent coins (while having unlimited numbers of other coins) and the `magic_table` as computed from `dp_cc`, you are to write an $O(1)$ time and space Python program to output the number of ways to form a given `amount` ($\leq$ \$1).

[3 marks]

```
def cc_5cent_limited_to(num, magic_table, amount):
```

C. BONUS (Not Easy; you don't have to do this if you have no time)

If you know how to do Part (B), then you should know how to write a similar function `cc_10cent_limited_to(num, magic_table, amount)`. With these two functions, you can use them to write an $O(1)$ time and space function to obtain the number of ways to form an amount ($\leq$ \$1), with only `num1` number of 5 cents coins, `num2` number of 10 cents coins, and unlimited number of 1 cent, 20 cents, and 50 cents coins. For example, with one 5 cents and two 10 cents coins and unlimited of the other, the needed function `cc_5cent_10cent_limited_to(1, 2, magic_table, 100)` outputs 46 ways to change \$1. [3 bonus marks]

— E N D O F P A P E R —