15-112
Fall 2023 Exam 2
November 5, 2023

Name:

Andrew ID:

- You may not use any books, notes, or electronic devices during this exam.
- You may not ask questions about the exam except for language clarifications.
- Show your work on the exam to receive credit.
- You may use the backs of pages as scratch paper.
- All code samples run without crashing except otherwise specified. Assume any imports are already included as required.
- You may assume that `random`, `math`, `string`, `cmu_graphics` and `copy` are imported; do not import any other modules.
- If you need extra space, use the back of the pages. Do not tear off any page. Doing so may be considered an academic integrity violation.

Don't write anything in the table below.

Question	Points	Score	Time
1	12		9
2	16		8
3	10		12
4	8		6
5	15		11
6	12		9
7	15		11
8	12		9
Total:	100		75

There are 100 points on this exam, lasting 75 minutes.
You can find *estimated* completion times for each question in the table above.

1. **Short answers**

   Answer each of the following *very briefly*.

   (a) (1 point) In 15-112, is $\mathcal{O}(N^2)$ equal to $\mathcal{O}(2N)$? Answer YES or NO in the box below.

   (b) (1 point) In 15-112, do we care about lesser-order terms, so $\mathcal{O}(N^2)$ is better than $\mathcal{O}(N^2 + 10000N + 100)$? Answer YES or NO in the box below.

   (c) (2 points) In one sentence, explain why this code will crash:

```
d = {'a', 'b', 'd'}
d['a'] = d['b'] + d['c']
```

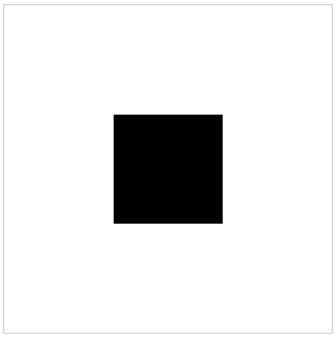
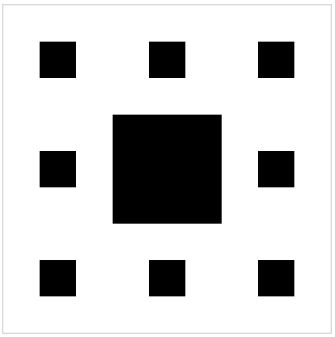
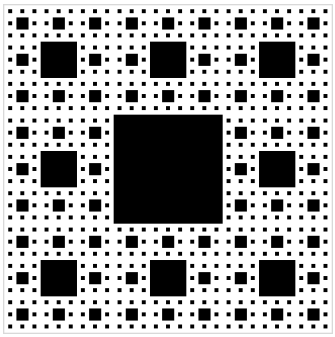
   (d) (2 points) The following function is meant to remove spaces within a string recursively. Indicate whether the function is correct or not. If not, explain what's wrong with it.

```
def recRemoveSpaces(s):
    res = ""
    if len(s) > 0:
        res = s[0]
    return res + recRemoveSpaces(s[1:])
```

(e) (2 points) The following code is a failed attempt to implement an animation. Ignoring the possible run-time/syntax errors, can you identify MVC violations? Write down the **line numbers** where MVC violations occur in the box below the code.

```
1  from cmu_graphics import *
2  import random
3
4  def onAppStart(app):
5      app.grid = [[0,0],[0,0]]
6      app.r = 20
7      app.cx, app.cy = app.width//2, app.height//2
8
9  def onMousePress(app, x, y):
10     dx = 0
11     dy = 0
12     if x > app.width // 2:
13         dx += 1
14     if y > app.height // 2:
15         dy += 1
16     distance = ((app.cx - x)**2 + (app.cy - y)**2)**0.5
17     if distance < app.r:
18         drawLabel("Hit!", app.cx, app.cy)
19         app.grid[dx][dy] += 1
20
21 def redrawAll(app):
22     app.cx = random.randint(0,app.width)
23     app.cy = random.randint(0,app.height)
24     drawCircle(app.cx, app.cy, app.r, fill="blue")
25
26 runApp(width=800, height=800)
```

(f) (4 points) Consider the following fractal drawn at levels 0, 1, and 3. Assume that the function `drawSierpinskiCarpet` is called to draw the fractal. In the implementation, how many calls to `drawSierpinskiCarpet` and `drawRect` occur in the base case? What about the recursive case?

Base Case		**Recursive Case**	
Number of calls to `drawSierpinskiCarpet`		Number of calls to `drawSierpinskiCarpet`	
Number of calls to `drawRect`		Number of calls to `drawRect`	

2. **Code Tracing**

Indicate what each will print. Place your answer (and nothing else) in the box next to or below each block of code.

(a) (6 points) CT1

```
def ct1(L):
    s = set()
    for v in L:
        if isinstance(v, dict):
            for k in v:
                s.add(v[k])
        else:
            for e in v:
                if e in s:
                    s.remove(e)
        print(s)
    L = L + [s]
    return L

M = ['CB',
     {2:'A','B':3,4:'C'},
     'BC3',
     [4, 3, 2]]
ct1(M)
print(M)
```

(b) (5 points) CT2

```
import copy
def ct2(L, A, n):
    A[-1][0] += n
    A[n%2] += [10*n]
    L.append(n)
    print(f'A: {A}')

L = [[3], [4]]
C = copy.copy(L)
D = copy.deepcopy(L)
ct2(L, C, 1)
ct2(L, D, 2)
print(f'L: {L}')
```

(c) (5 points) CT3

```
def ct3Helper(n, m):
    if n < 10:
        return m*10 + n

    if (n%10)%2 == 0:
        m = m * 10 + n%10
    print(f'in {n}  {m}')
    res = ct3Helper(n//10, m)
    print(f'out {res}')
    return res

def ct3(n):
    return ct3Helper(n, 0)

print(ct3(2143))
```

3. **Reasoning Over Code**

For each function, find values of the parameters so that the following functions will return `True`. Place your answer (and nothing else) in the box below each code.

(a) (5 points) ROC1: Indicate the value of `L` for the function `roc1` return `True`. Make sure strings are enclosed with quotes and lists with brackets.

```
def roc1(L):
    assert(len(L) == 3)
    count = 0
    for row in L:
        assert(len(row) == 3)
        for i in range(1,len(row)):
            if count % 2 == 0:
                assert(sum(row) == 18)
                assert(row[i-1] < row[i])
            else:
                assert(sum(row) == 15)
                assert(row[i] < row[i-1])
        count += 1
    return True
```

(b) (5 points) ROC2: Indicate the value of `s` for the function `roc2(s)` return `True`. Note: `rocHelper2(s, i, d)` is a *helper* function, and your answer should ONLY refer to `roc2(s)`.

```
def rocHelper(s, i, d):
    m = len(s)//2
    if d == 0:
        if i % 2 == 0:
            return s == 'A'
        else:
            return s == 'B'
    elif len(s) % 2 == 0:
        return rocHelper(s[:m], i+1, d-1) and rocHelper(s[m:], i+2, d-1)
    else:
        return s[m] == str(d) and rocHelper(s[:m], i+1, d-1) \
                                  and rocHelper(s[m+1:], i+2, d-1)

def roc2(s):
    assert(len(s) % 2 == 1)
    return rocHelper(s, 0, 2)
```

4. (8 points) **Big-O**

For each function shown below, write next to each line of the function, either the Big-O runtime of the line or the number of times the line loops. Then, write the total Big-O runtime of the function in terms of N in the box to the right of the code. **All answers must be simplified- do not include lower-order terms! For full credit, you must include line-by-line Big-O.**

```
def bigOh1(L): # L is a NxN 2D list
    N = len(L)                          _____
    s = 0                               _____
    for row in L:                       _____
        a = sorted(row)                 _____
        s += a[0]                       _____
    return s                            _____
```

```
def bigOh2(L): # L is a list with N elements
    newL = []                           _____
    n = len(L)                          _____
    for i in range(0, n, n//4):         _____
        newL.append(L.pop())            _____
    return newL                         _____
```

```
def bigOh3(s):  # s is a string with N characters
    if s.isdigit():                     _____
        letters = set(s)                _____
        for c in letters:               _____
            if c == '4':                _____
                return "YES"            _____
    return "NO"                         _____
```

```
def bigOh4(L): # L is a list with N elements
    d = dict()                          _____
    for value in L[::-1]:               _____
        d[value] = d.get(value, 0) + 1  _____
    return d                            _____
```

5. (15 points) **Free Response: isValidFourNeighborColoring**

Write the function `isValidFourNeighborColoring(L)` that takes an arbitrary value `L` and returns `True` if `L` satisfies the criteria of a *four neighbor coloring rectangle*, and `False` otherwise. The following conditions define a *four neighbor coloring rectangle*:

- L is a non-empty 2D rectangular list.
- L consists exclusively of single-character strings representing colors, which can be `'G'` (green), `'R'` (red), `'Y'` (yellow), and `'B'` (blue).
- No two adjacent cells within the rectangular list `L` have the same color. Here, adjacent cells are defined as those located in the up, down, left, and right directions surrounding a given cell.

Here are some test cases:

```
L1 = [['Y', 'G', 'R', 'B'],
      ['B', 'Y', 'B', 'Y'],
      ['R', 'G', 'R', 'G'],
      ['B', 'Y', 'B', 'Y']]

L2 = [['B', 'Y', 'B', 'Y']]

L3 = [['R', 'G', 'B', 'G'],
      ['B', 'Y', 'B', 'Y'],
      ['R', 'G', 'R', 'G'],
      ['B', 'Y', 'B', 'Y']]

L4 = [['R', 'G', 'R', 'G'],
      ['B', 'Y', 'B'],
      ['R', 'G', 'R', 'G'],
      ['B', 'Y', 'B', 'Y']]

L5 = [['RED', 'G', 'RED', 'G'],
      ['B', 'Y', 'B', 'Y'],
      ['RED', 'G', 'RED', 'G'],
      ['B', 'Y', 'B', 'Y']]

L6 = "Remember the Magic Square?"

assert(isValidFourNeighborColoring(L1) == True)  # OK
assert(isValidFourNeighborColoring(L2) == True)  # Also OK
assert(isValidFourNeighborColoring(L3) == False) # L[0][2] == L[1][2]
assert(isValidFourNeighborColoring(L4) == False) # Not rectangular
assert(isValidFourNeighborColoring(L5) == False) # Invalid value (RED)
assert(isValidFourNeighborColoring(L6) == False) # Not even a list
```

Answer space for Question 5

6. (12 points) **Free Response: organizeRosters**

You are given a 2D list `studentData`, where each row contains precisely three `string` elements (a program name, a course number, and a name). The following is an example value for `studentData`:

```
studentData = [('CS','15-213','ahmed'),
               ('IS','67-262', 'betty'),
               ('IS','67-262', 'ahmed'),
               ('CS','15-150','ahmed'),
               ('HCI','05-318','fred'),
               ('BS','70-340','huda'),
               ('BS','70-340','omar'),
               ('BS','70-257', 'ahmed'),
               ('BS','70-257', 'betty')]
```

Write the function `organizeRosters(studentData)` that takes data formatted like the list shown above and returns a dictionary mapping each program to another dictionary that maps each course number to a set of the names in the data for that course number.

For example, for the `studentData` given above, `organizeRosters(studentData)` returns:

```
{'CS':
     {'15-213': {'ahmed'},
      '15-150': {'ahmed'}
     },
 'IS':
     {'67-262': {'betty', 'ahmed'}
     },
 'HCI':
     {'05-318': {'fred'}
     },
 'BS':
     {'70-340': {'omar', 'huda'},
      '70-257': {'ahmed', 'betty'}
     }
}
```

Your code should work with any data, including programs, courses, and names not given in the example, so do not hardcode! You can assume that the value of `studentData` is correctly set as described above.

Additional Space for Answer to Question 6

7. (15 points) **Free Response: longestScheduleMatch**

You are given a dictionary `rosters` formatted like the return value of the previous question: a dictionary mapping each program to another dictionary that maps each course number to a set of the names in the data for that course number. Your task is to identify the longest matching schedule. A matching schedule occurs when two or more students are enrolled in the exact same set of courses. Write the function `longestScheduleMatch(rosters)` that takes a dictionary `rosters` as described above and returns the size of the longest matching schedule. For instance, for

```
roster1 = {
 'CS':
     {'15-213': {'ahmed'},
      '15-150': {'ahmed'}
     },
 'IS':
     {'67-262': {'betty', 'ahmed'}
     },
 'HCI':
     {'05-318': {'fred'}},
 'BS':
     {'70-340': {'omar', 'huda'},
      '70-257': {'ahmed', 'betty'}
     }
}
```

`longestScheduleMatch(roster1)` returns 1 because Omar and Huda have the only matching schedule of one course (70-340). Here's another test case:

```
roster2 = {
 'IS':
     {'67-262': {'betty', 'ahmed'}
     },
 'CS':
     {'15-150': {'ahmed', 'betty'}
     },
 'BA':
     {'09-105': {'fred'}
     },
 'BS':
     {'70-340': {'omar', 'huda'},
      '70-257': {'ahmed', 'betty'}
     }
}
```

`longestScheduleMatch(roster2)` returns 3 because Ahmed and Betty have a matching schedule of three courses: 67-262, 15-150, and 70-257.

If there are no matching schedules, the function should return 0.

NOTE: For full credit, your function should run in $\mathcal{O}(N)$ time, where N is equal to the number of students. You can assume that the number of programs and courses is constant.

Additional Space for Answer to Question 7

8. (12 points) **Free Response: Recursive Rearrange Letters**

   Write the **recursive** function `recRearrangeLettersInFront(s)`, which takes a string `s` (which may contain any valid ASCII characters). The function should return a new string with all the alphabetical letter characters moved to the front, maintaining their original order. The non-alphabetical characters should remain at the end of the result string, preserving their original relative order as in the input string `s`.

   For instance,

```
assert(recRearrangeLettersInFront('teatime4me') == 'teatimeme4')
assert(recRearrangeLettersInFront('h3lp3rfunct10n5') == 'hlprfunctn33105')
assert(recRearrangeLettersInFront(':-soeasy)') == 'soeasy:-)')
assert(recRearrangeLettersInFront('1Ch34?e\te&7') == 'Chee134?\t&7')
assert(recRearrangeLettersInFront('ILove15-112') == 'ILove15-112')
assert(recRearrangeLettersInFront('123') == '123')
assert(recRearrangeLettersInFront('') == '')
```

   Your solution must use recursion. If you use any loops or iterative functions, you will receive no points on this problem.

Additional Space for Answer to Question 8