Eric Roberts and Mark Hopkins
CSCI 121

Handout #61
December 3, 2018

Practice Final Examination #1

Review session: **Friday, December 7, 3:10–4:30P.M. (Psychology 105)**
Scheduled final: **Monday, December 10, 1:00–4:00P.M. (Psychology 105)**

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the final examination. A solution set to this practice examination will be handed out on Wednesday along with a second practice exam.

Time of the exam

The final exam is scheduled for Monday, December 10, from 1:00–4:00P.M. in the regular classroom. If you are unable to take the exam at the scheduled time or if you need special accommodations, please send an email message to `esroberts@reed.edu` stating the following:

- The reason you cannot take the exam at the scheduled time.
- A list of 3-hour blocks (or longer if you have special accommodations) on Monday, Tuesday, or Wednesday of exam week at which you could take the exam. These time blocks must be during the regular working day and must therefore start between 8:30 and 2:00.

In order to arrange special accommodations, we must receive a message from you by 5:00P.M. on Thursday, December 6. Replies will be sent by electronic mail on Friday, December 7.

Review session

There will a review session on Friday, December 7, from 3:10–4:30P.M. in the regular classroom (Psychology 105). We will announce the winners of the Adventure Contest and hold the random grand-prize drawing at the beginning of the review session.

Coverage

The exam covers the material presented in the textbook, along with the discussion of recursion in Handout #30.

General instructions

The instructions that will be used for the actual final look like this:

> Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.
>
> Each question is marked with the number of points assigned to that problem. The total number of points is 100. We intend that the number of points be roughly equivalent to the number of minutes someone who is completely on top of the material would spend on that problem. Even so, we realize that some of you will still feel time pressure. If you find yourself spending a lot more time on a question than its point value suggests, you might move on to another question to make sure that you don't run out of time before you've had a chance to work on all of them.
>
> In all questions, you may include functions or definitions that have been developed in the course, either by writing the **`import`** line for the appropriate package or by giving the name of the function and the handout or chapter number in which that definition appears.
>
> The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Note: To conserve trees, we have cut back on answer space for the practice exams. The actual final will have much more room for your answers and for any scratch work.

Please remember that the final is <u>open-book.</u>

Problem 1—Short answer (10 points)

1a) Suppose that the list **`array`** has been initialized as follows:

```
array = [ 10, 20, 30, 40, 50 ]
```

This declaration sets up a list of five elements with the initial values shown in the diagram below:

`array`

10	20	30	40	50

Given this array, what is the effect of calling the function

```
mystery(array)
```

if **`mystery`** is defined as:

```
def mystery(list):
    tmp = list[-1]
    for i in range(len(list)):
        list[i] = list[i - 1]
    list[0] = tmp
```

Work through the function carefully and indicate your answer by filling in the boxes below to show the final contents of **`array`**:

`array`

1b) What is the result of calling the function **`enigma`** in the following code:

```
def enigma():
    def puzzle(x):
        def riddle(y):
            return 2 * x - y
        return riddle
    x = 10
    y = 37
    f = puzzle(y)
    print(f(x))
```

Problem 2—Simple graphics (15 points)

The Portable Graphics Library fills a `GArc` by filling the wedge-shaped region formed by connecting the ends of the arc to the center. Although this definition is not what you want for all applications, it turns out to be perfect for the problem of displaying a traditional pie chart. Your job in this problem is to write a function

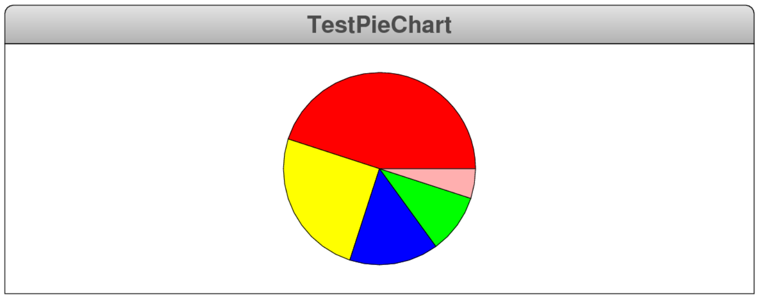
```
def createPieChart(r, data):
```

that creates a `GCompound` object for a pie chart with a set of data values, where `r` represents the radius of the circle, and `data` is the array of data values you want to plot.

The operation of the `createPieChart` function is easiest to illustrate by example. If you execute the following function:

```
def TestPieChart()
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    data = [ 45, 25, 15, 10, 5 ]
    pieChart = createPieChart(50, data)
    gw.add(pieChart, gw.getWidth() / 2, gw.getHeight() / 2)
```

your program should generate the following pie chart in the center of the window:



The red wedge corresponds to the 45 in the data array and extends counterclockwise through 45% of the circle, which is not quite halfway. The yellow wedge then picks up where the red wedge left off and extends for 25% of a complete circle. The blue wedge takes up 15%, the green wedge takes up 10%, and the pink wedge the remaining 5%.

As you write your solution to this problem, you should keep the following points in mind:

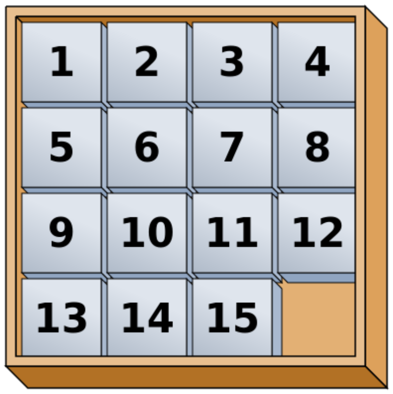
- The values in the array are not necessarily percentages. What you need to do in your implementation is to divide each data value by the sum of the elements to determine what fraction of the complete circle each value represents.
- The colors of each wedge are specified in the following constant array:

  ```
  WEDGE_COLORS = [
      "Red", "Yellow", "Blue", "Green", "Pink", "Cyan"
  ]
  ```

  If you have more wedges than colors, you should just start the sequence over, so that the seventh wedge would be red, the eighth yellow, and so on.
- The reference point of the `GCompound` returned by `createPieChart` must be the center of the circle.

Problem 3—Interactive graphics (15 points)

In all probability, you have at some point seen the classic "Fifteen Puzzle" which first appeared in the 1880s. The puzzle consists of 15 numbered squares in a 4×4 box that looks like the following picture, which is taken from the Wikipedia entry for the puzzle:



One of the squares is missing from the 4×4 grid. The puzzle is constructed so that you can slide any of the adjacent squares into the position taken up by the missing square. The object of the game is to restore a scrambled puzzle to its original state.

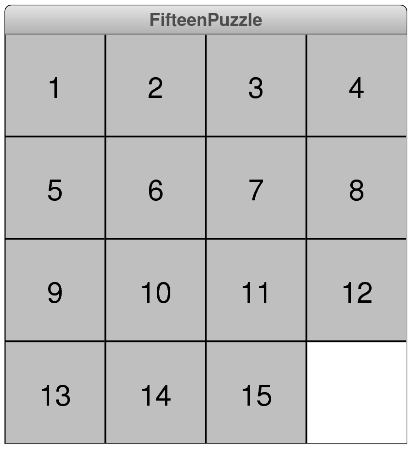
Your task is to simulate the Fifteen Puzzle, which is easiest to do in two steps:

Step 1.

Write a program **`FifteenPuzzle`** that displays the initial state of the Fifteen Puzzle with the 15 numbered squares arranged as shown in the diagram. Each of the pieces should be a **`GCompound`** containing a square filled in light gray, with a number centered in the square using an 18-point Sans-Serif font, as specified by the following constants:

```
SQUARE_SIZE = 60
GWINDOW_WIDTH = 4 * SQUARE_SIZE
GWINDOW_HEIGHT = 4 * SQUARE_SIZE
SQUARE_FILL_COLOR = "LightGray"
PUZZLE_FONT = "18px 'SansSerif'"
```

When you have finished the code for step 1, the graphics window should look like this:

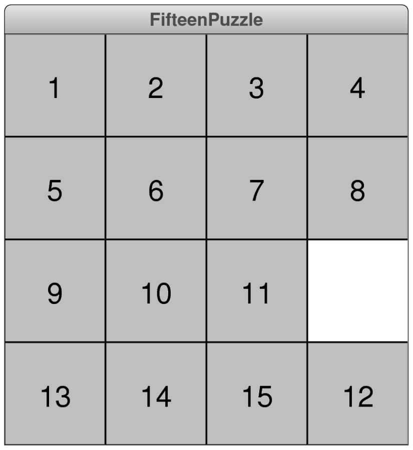


Step 2.

Animate the program so that clicking on a square moves it into the adjacent empty space, if possible. This task is easier than it sounds. All you need to do is:

1. Figure out which square you clicked on, if any, by using **`getElementAt`** to check for an object at that location.
2. Check the adjacent squares to the north, south, east, and west. If any square is inside the window and unoccupied, move the square in that direction. If none of the directions work, do nothing.

For example, if you click on the square numbered 5 in the starting configuration, nothing should happen because all of the directions from square 5 are either occupied or outside of the window. If, however, you click on square 12, your program should figure out that there is no object to the south and then move the square to that position, as follows:



Problem 4—Strings (15 points)

In Dan Brown's best-selling novel *The Da Vinci Code,* the first clue in a long chain of puzzles is a cryptic message left by the dying curator of the Louvre. Two of the lines of that message are

O, Draconian devil!
Oh, lame saint!

Professor Robert Langdon (the hero of the book, played by Tom Hanks in the movie) soon recognizes that these lines are ***anagrams***—pairs of strings that contain exactly the same letters even if those letters are rearranged—for

Leonardo da Vinci
The Mona Lisa

Your job in this problem is to write a predicate function

```
isAnagram(s1, s2)
```

that takes two strings and returns `True` if they contain exactly the same alphabetic characters, even though those characters may appear in any order. Thus, your function should return `True` for each of the following calls:

```
isAnagram("O, Draconian devil!", "Leonardo da Vinci")
isAnagram("Oh, lame saint!", "The Mona Lisa")
isAnagram("ALGORITHMICALLY", "logarithmically")
isAnagram("Doctor Who", "Torchwood")
```

These examples illustrate two important requirements for the `isAnagram` function:

- The implementation should look only at letters, ignoring any extraneous spaces and punctuation marks thrown in along the way.
- The implementation should ignore the case of the letters in both strings.

There are many different algorithmic strategies you could use to decide whether two strings contain the same alphabetic characters. If you're having trouble coming up with a strategy, you should note that two strings are anagrams if and only if they generate the same letter-frequency table, as illustrated in the `CountLetterFrequencies` program on page 240 of the text.

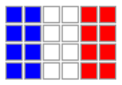
Problem 5—Arrays (10 points)

Write a function

```
def doubleImage(oldImage)
```

that takes an existing **`GImage`** and returns a new **`GImage`** that is twice as large in each dimension as the original. Each pixel in the old image should be mapped into the new image as a 2×2 square in the new image where each of the pixels in that square matches the original one.

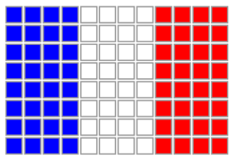
As an example, suppose that you have a **`GImage`** from the file **`TinyFrenchFlag.png`** that looks like this, where the diagram has been expanded so that you can see the individual pixels, each of which appears as a small outlined square:



This 6×4 rectangle has two columns of blue pixels, two columns of white pixels, and two columns of red pixels. Calling

```
biggerFrenchFlag = doubleImage(GImage("TinyFrenchFlag.png"))
```

should create a new image with the following 12×8 pixel array:



The blue pixel in the upper left corner of the original has become a square of four blue pixels, the pixel to its right has become the next 2×2 square of blue pixels, and so on.

Keep in mind that your goal is to write an implementation of **`doubleImage`** that works with any **`GImage`** and not just the flag image used in this example.

Problem 6—Recursive functions (10 points)

The following iterative function raises **x** to the **k**th power for any nonnegative integer **k**:

```
def raiseToPower(x, k):
    result = 1
    for i in range(k):
        result *= x
    return result
```

Rewrite this function so that it operates recursively, taking advantage of the following insight:

- If **k** is even, x^k is the square of **x** raised to the power **k // 2**.
- If **k** is odd, x^k is the square of **x** raised to the power **k // 2** times **x**.

In solving this problem, you need to identify the simple cases necessary to complete the recursive definition. You must also make sure that your code is efficient in the sense that it makes only one recursive call per level of the recursive decomposition. Finally, your code must perform all the computation necessary to produce the value of the result and may not use the ** operator or the functions in the **math** class that implement exponents.

Problem 7—Defining classes (10 points)

Figure 1 presents some code for a `Car` class. Write the code for a new class, `Delivery`, used to represent delivery trucks for a particular company. Delivery trucks are like cars, but rather than allowing for customization, these trucks should always have a gas tank that holds 50 gallons and get 15 miles per gallon. Furthermore, there should be an additional method, `idle`, which takes as input a time measured in minutes and represents the truck idling at a stop for that many minutes. The truck, when idling, burns two gallons per hour. (If a truck is asked to idle for long enough to burn more gas than there is in the tank, the tank just becomes emptied.)

[Note: This problem appeared on this year's Junior Qual for computer science]

Figure 1. The `Car` class

```
from math import sqrt

class Car:
    def __init__(self, mpg, tankSize):
        self.currentX = 0            # starts at the origin
        self.currentY = 0
        self.mpg = mpg
        self.tankSize = tankSize
        self.currentGas = tankSize    # starts with full tank

    def moveTo(self, newX, newY):
        distance = sqrt((self.currentX-newX)**2 + (self.currentY-newY)**2)
        if self.currentGas < distance/self.mpg:
            return False
        else:
            self.currentGas -= distance/self.mpg
            self.currentX = newX
            self.currentY = newY
            return True
```

Problem 8—Python data structures (15 points)

Adventure was not the first widely played computer game in which an adventurer wandered in an underground cave. As far as we know, that honor belongs to the game "Hunt the Wumpus," which was developed by Gregory Yob in 1972.

In the game, the wumpus is a fearsome beast that lives in an underground cave composed of 20 rooms, each of which is numbered between 1 and 20. Each of the twenty rooms has connections to three other rooms, represented as a list—one element per room—of three-element lists containing the numbers of the connecting rooms. (Because the room numbers start with 1 instead of 0, the data structure stores some arbitrary value in element 0 of the outer list.) In addition to the connections, the Python dictionary that stores the data for the wumpus game also keeps track of the room number occupied by the player and the room number in which the wumpus resides.

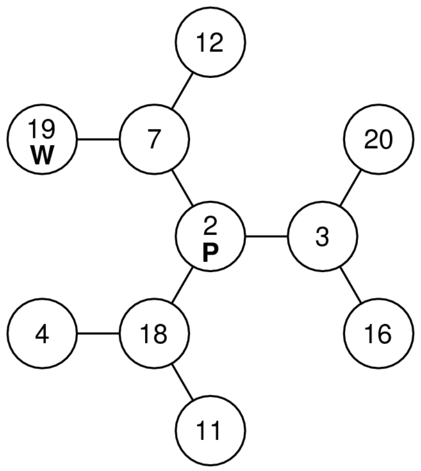
In an actual implementation of the wumpus game, the information in this data structure would be generated randomly. For this problem, which focuses on whether you can work with data structures that have already been initialized, you can imagine that the variable **`WUMPUS_CAVE`** has been initialized to the dictionary shown in Figure 2, which conforms to the syntactic rules of both Python and JSON. The data structure shows the following:

- The player is in room 2
- The wumpus is in room 19
- Room 1 connects to rooms 6, 14, and 19; room 2 connects to 3, 7, and 18; and so on.

Figure 2. Python/JSON representation of the data structure for the wumpus cave

```
{
    "playerLocation": 2,
    "wumpusLocation": 19,
    "connections": [
        -1,
        [6, 14, 16],
        [3, 7, 18],
        [2, 16, 20],
        [6, 18, 19],
        [8, 9, 11],
        [1, 4, 15],
        [2, 12, 19],
        [5, 10, 13],
        [5, 11, 17],
        [8, 14, 16],
        [5, 9, 18],
        [7, 14, 15],
        [8, 15, 20],
        [1, 10, 12],
        [6, 12, 13],
        [1, 3, 10],
        [9, 19, 20],
        [2, 4, 11],
        [4, 7, 17],
        [3, 13, 17]
    ]
}
```

To help you visualize the situation, here is a piece of the cave map centered on the location of the player in room 2:



The player is in room 2, which has connections to rooms 3, 7, and 18. Similarly room 7 has connections to rooms 2, 12, and 19, which is where the wumpus is lurking. The other connections from rooms 4, 11, 16, 20, 12, and 19 are not shown.

It was usually possible to avoid the wumpus because the wumpus was so stinky that the player could smell the wumpus from up to two rooms away. Thus, in the diagram above, the player can smell the wumpus. If, however, the wumpus were to wake up and move to a room beyond the boundaries of this diagram, the scent of the wumpus would disappear.

Write a predicate function **`doesPlayerSmellTheWumpus`**, which takes the entire data structure as an argument and returns **`True`** if the player smells the wumpus and **`False`** otherwise. Thus, calling

```
doesPlayerSmellTheWumpus(WUMPUS_CAVE)
```

would return **`True`**. The function would also return **`True`** if the wumpus were in rooms 3, 7, or 18, which are one room away from the player. If, however, the wumpus were in one of the rooms not shown on this map, **`doesPlayerSmellTheWumpus`** would return **`False`**.