Practice Final Examination #1

Review session: Sunday, December 15, 7:00–8:30 P.M. (Psychology 105)
Scheduled final: Wednesday, December 18, 9:00 A.M.–12:00 NOON (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 Friday along with a second practice exam.

Time of the exam

The final exam is scheduled for Wednesday, December 18, from 9:00 A.M.–12:00 NOON 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 5:00 P.M. on Wednesday, December 11. Replies will be sent by electronic mail by the end of that week.

Review session

There will be a review session on Sunday, December 15, from 7:00–8:30 P.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.

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.

**Problem 1—Short answer (10 points)**

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

```python
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:

```python
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:

```python
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 (10 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

```python
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:

```python
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:

```java
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:

![Fifteen Puzzle](image)

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:

```java
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:

![FifteenPuzzle](image)

**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:

![FifteenPuzzle](image)

**Problem 4—Strings (10 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

```python
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:

```python
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 246 of the text.

**Problem 5—Arrays (10 points)**

Write a function

```python
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

```python
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)**

One of the examples from the discussion of recursion was the Koch fractal, which produces snowflake patterns like these:

```
order 0  
order 1  
order 2  
order 3  
```

Each new order is formed by replacing every straight line segment with a sequence of four line segments that looks like this:

where each of the new segments is one-third the size of the original. The total length of the fractal line is therefore four-thirds of the original edge length. The process of increasing the length of each edge means that the perimeter (that is, the total length of all the segments on the border) keeps increasing as the order grows.

Your job in this problem is to write a recursive function

```python
def snowflakePerimeter(edge, order):
```
that returns the perimeter of a fractal snowflake whose order-0 edge length is given by the parameter \( \text{edge} \) and whose level of fractal subdivision is given by \( \text{order} \). For example, calling \( \text{snowflakePerimeter}(1, 0) \) should return 3. Moreover, because each line around the edge is replaced by a new fractal line that is four-thirds times the original length, calling \( \text{snowflakePerimeter}(1, 1) \) should return 4.

In writing your answer, you should keep the following points in mind:

- You don’t have to do any graphics. The solution to this problem uses only recursion and numeric calculation.
- Your function must operate recursively using a structure that mirrors the decomposition of the fractal.

Problem 7—Defining classes (10 points)

In the Adventure project, I chose to represent passages as tuples mostly to give you some experience with one of Python’s most widely used data types. One can make a strong argument that a better strategy would be to define a new class called \( \text{AdvPassage} \) that supports the following methods:

- A constructor that takes a string of the form found in the rooms data file. For example, calling \( \text{AdvPassage}("\text{DOWN: BeneathGrate/KEYS}\)" \) should create a new \( \text{AdvPassage} \) object that represents moving downward to the room named "BeneathGrate", but that passage is only usable if the player is carrying the object named "KEYS". As in the Adventure project, the third component of a passage string need not be present, so that calling \( \text{AdvPassage}("\text{IN: InsideBuilding}\)" \) creates a passage that always takes the player to the room named "InsideBuilding" in response to the motion verb "IN".

- The getter methods \( \text{getDirection} \), \( \text{getDestination} \), and \( \text{getKey} \), each of which returns the value of the specified component of the \( \text{AdvPassage} \) object. For example, calling the \( \text{getDirection} \) method on the first object described in the previous bullet point would return the string "DOWN". Calling \( \text{getKey} \) on the second example should return the value \( \text{None} \).

Write a complete class definition for the \( \text{AdvPassage} \) class.

Problem 8—Linked structures (10 points)

The code for the linked-list queue presented in class appears in Figure 1 on the next page. Add a new method called \( \text{priorityEnqueue} \) that takes a value and adds that value to the head of the queue, putting it ahead of all the other items.

As an example, suppose that you execute the following code after adding this extension:

```python
q = Queue()
q.enqueue("A")
q.enqueue("B")
q.priorityEnqueue("C")
```

calling \( q.\text{dequeue} \) three times should return the strings "C", "A", and "B", in that order.
Problem 9—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.

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

![Cave Map](image)

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

```python
{
    "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]
    ]
}
```
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 \texttt{doesPlayerSmellTheWumpus}, which takes the entire data structure as an argument and returns \texttt{True} if the player smells the wumpus and \texttt{False} otherwise. Thus, calling

\begin{verbatim}
doesPlayerSmellTheWumpus(WUMPUS_CAVE)
\end{verbatim}

would return \texttt{True}. The function would also return \texttt{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, \texttt{doesPlayerSmellTheWumpus} would return \texttt{False}. 