CSCI 121: Practice Final Exam

Review session: 7-9pm, Tuesday, December 13th, in Eliot 314
Exam: 1pm-5pm, Thursday, December 15th, VLH

Fall 2022

The next pages give practice problems for the final exam being held next week. The exam is comprehensive and covers these topics:

- scripting with `input` and `print`
- variables and assignment
- integer arithmetic, boolean connectives, integer comparisons
- strings and string operations
- integer division using `%` and `//`
- printing versus returning, the `None` type
- conditional statements and loops
- function definitions
- recursive functions
- higher-order functions and `lambda`
- Python's management of variable frames
- lists and dictionaries
- object-orientation and inheritance
- linked lists and binary search trees
- sorting and searching

You can use these to test your knowledge in preparation for taking the exam. I will post my solutions to these problems on Saturday and can go over my solutions in the review session.

1. Write a Python function `justEvens(someList)` that takes a list of integers and returns a *new list* containing only its even elements, and in the same order they appeared in the list. For example:

```
>>> justEvens([1,2,3,4,5])
[2, 4]
>>> justEvens([1,2,3,4,2,2,3,2,1,1,6])
[2, 4, 2, 2, 2, 6]
>>> justEvens([5,3,1])
[]
```

2. The sum of the first five squares is

$$1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$$

Write a **recursive** function `sumSquares(n)` that computes and returns the sum of the first `n` squares. For example:

```
>>> sumSquares(5)
55
>>> sumSquares(2)
5
>>> sumSquares(1)
1
```

Your function **must be recursive.** It can assume that `n` is a positive number.

3. Note that:

   - The number 9 has three positive divisors, 1, 3, and 9.
   - The number 10 has four positive divisors, 1, 2, 5, and 10.
   - The number 11 has only two positive divisors, 1 and 11.
   - And the number 12 has 6 positive divisors, 1, 2, 3, 4, 6, and 12.

   Write a Python function `mostDivisors(start,end)` that determines the largest number of positive divisors among any the numbers from `start` up to and including `end`. It should return that number of divisors. For example:

```
>>> mostDivisors(9,12) # because of 12
6
>>> mostDivisors(9,11) # because of 10
4
>>> mostDivisors(9,9)
3
```

   You can assume both parameters are positive and `end` is greater than or equal to `start`.

4. Below is the definition of two classes for a linked list, similar to we wrote in lecture:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.first = None
    def prepend(self, value):
        newNode = Node(value)
        newNode.next = self.first
        self.first = newNode
    def output(self):
        current = self.first
        while current is not None:
            print(current.value)
            current = current.next
```

Write a method `appendSeveral` that appends a value some specified number of times to the end of a linked list. For example:

```
>>> ll = LinkedList()
>>> ll.prepend(3)
>>> ll.prepend(1)
>>> ll.prepend(8) # Places 8 at the front, with 1 then 3 following.
>>> ll.output()
8
1
3
>>> ll.appendSeveral(7,3) # Places three 7s at the end.
>>> ll.output()
8
1
3
7
7
7
```

5. Write a function `pairQuery` that **returns a function** back. It takes a two integers as parameters. The function it gives back can be used to obtain each of the integers it was given using the strings `"first"` and `"second"`. Here is an example of its use:

```
>>> pq = pairQuery(89,333)
>>> pq("first")
89
>>> pq("second")
333
>>> another = pairQuery(18,2)
>>> another("first")
18
>>> another("second")
2
>>> x = another("second")
>>> x
2
```

Note (as suggested by the last interaction using `x`) that `pq` and `another` don't `print` values. Instead, they `return` one or the other of their pair, depending on what string they are given.

You can assume that a query function returned by `pairQuery` will only be asked for `"first"` or for `"second"`, and never any other string.

6. Let's invent a dispenser object that contains an array of liquids, each liquid with its own name. It holds up to a liter of each liquid that it dispenses, and all the liquid containers are initially empty. Each liquid has a string describing it. When it is asked to `dispense` some liquid it holds, a particular fraction of a liter of that liquid is dispensed. For example:

   ```
   >>> d = Dispenser(["shampoo", "conditioner"], 0.8)
   ```

   With the above, `d` holds shampoo and conditioner. It dispenses 0.8 liters (!) of whatever liquid is requested.

   A `Dispenser` is built by giving the `liquids` as a list of strings, and a `dispenseAmount` as a fraction of a liter. Also:

   - It has a method `refill` that leads to each container being filled up so it has a liter of each liquid.
   - It has a method `amounts` that outputs all the liquids with the amount of each.
   - The `dispense` method decreases the specified liquid by the `dispenseAmount`, or instead by the amount of the liquid it held when less than the `dispenseAmount`. It returns how much of that liquid was dispensed.

   Below continues use of the `Dispenser` object `d`:

   ```
   >>> d.amounts()
   shampoo: 0.0
   conditioner: 0.0
   >>> d.dispense("conditioner")
   0.0
   >>> d.fill()
   >>> d.amounts()
   shampoo: 1.0
   conditioner: 1.0
   >>> d.dispense("shampoo")
   0.8
   >>> d.amounts()
   shampoo: 0.2
   conditioner: 1.0
   >>> d.dispense("shampoo")
   0.2
   >>> d.amounts()
   shampoo: 0.0
   conditioner: 1.0
   ```

   Write the code for the class `Dispenser` on this and the next page.

6. Your `Dispenser` code can be continued below here.

7. Now let's invent a `ShowerDispenser` object that inherits the behavior of `Dispenser`, but always contains only `"shampoo"` and `"conditioner"`. Also, it always dispenses 0.1 liters with each `dispense` call. Furthermore, when constructed, it is initially full of each liquid, rather than empty.

   It has two additional methods `shampoo` and `conditioner`.

   - The `shampoo` method dispenses shampoo twice in succession, and returns the total amount dispensed by those two `dispense` calls.
   - The `conditioner` method dispenses conditioner once, and returns the amount dispensed.

   Write the definition of `ShowerDispenser`. Your code should rely on the superclass `Dispenser` as best it can.

8. You can check whether something is a list using Python's `isinstance` like so:

```
>>> isinstance([1,2,3,4], list)
True
>>> isinstance(345, list)
False
>>> isinstance("6789", list)
False
>>> isinstance([], list)
True
```

Define an *integer nesting* to be a list whose elements are either integers or else also integer nestings. This means that nestings are lists of integers and lists, where those lists contain a mix of integers and lists, and so on. Here are some examples of integer nestings:

```
 [1, [2, 3], [4, [5, 6], 7], 8]
 [[1, 2], [3, 4], 5, [[6, 7], 8]]
[[1, [2], 3, [4, [5]], [[6, 7]], 8]
```

Write a function `nestingContains(nesting, value)` that takes a `nesting` and an integer `value` and returns `True` if `value` appears somewhere as an element of `nesting`. If `value` never occurs as an element of any list in `nesting`, the function should return `False`. For example:

```
>>> nestingContains([1,2,3],4)
False
>>> nestingContains([1,2,3],3)
True
>>> nestingContains([1,[2],3],2)
True
>>> nestingContains([1,[2],3],4)
False
>>> nestingContains([1,[2,[4,5]],3],5)
True
>>> nestingContains([1,[2,[4,5]],3],6)
False
```

Hint: the function `nestingContains` should probably be recursive.

9. You are given a sorted list of integers `sortedList` and another integer `value`. Write a function `contains(sortedList, value)` that returns `True` if `value` is among the list of values in `sortedList`.

   You must write the function "from scratch" using only basic list operations like `len` and notation like `sortedList[`*index*`]` to check for items in the list. In particular you can't use certain built-in Python operations like `in` that would make the coding trivial.

   Write the code so that it checks as few items in the list as is possible.

   (a) Write your code for `contains` below:

   (b) What is the running time of your code? Briefly explain why and give the running time using asymptotic notation. In doing so, you can use the variable n as the length of `sortedList`.

10. Below we give some code for a function `containsRepeat(someList)` that determines whether a value in `someList` appears more than once. Here is how it is supposed to work:

```
>>> containsRepeat([2,1,5,8,1,3]) # the value 1 is repeated
True
>>> containsRepeat([3,3,55,18,1,3]) # the value 3 is repeated
True
>>> containsRepeat([12,3,55,18,1]) # no repeats
False
```

Here is their code:

```
def containsRepeat(someList):
    i = 0
    while i < len(someList):
        j = 0
        while j < len(someList):
            if someList[i] == someList[j]:
                return True
            j += 1
        i += 1
    return False
```

(a) There is a bug in the code. It always returns `True`. What is the mistake in their code? Briefly give a fix that repairs their code.

(b) What is the running time of the code? Briefly explain why and then also give the running time using asymptotic notation. In doing so, you can use the variable n as the length of `someList`.

(c) Is there a more efficient strategy for checking for a repeated value in a list? Briefly sketch an algorithm (you don't need to write the code) that beats the running time of the code above. Give the running time of this improvement.