CSCI 121: Practice Final Exam

Exam: 1pm-4pm, Thursday, May 15th, Library 204

Spring 2025

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`
- lists and dictionaries
- object-orientation and inheritance
- linked lists and binary search trees
- sorting and searching, efficiency and running time
- file I/O and exceptions

You can use these to test your knowledge in preparation for taking the exam.

1. Write a Python function `justEvens` that takes a dictionary whose entries' keys are associated with integer values. It should return a list of the keys whose values are even. For example:

```
>>> justEvens({'a':1,'b':2,'c':3,'d':4,'e':5})
['b', 'd']
>>> justEvens({'z':1,'q':2,'w':3,'u':2,'a':2,'l':3,'c':2,'y':1,'k':6})
['q', 'u', 'a', 'c', 'k']
>>> justEvens({'a':5,'b':3,'e':1})
[]
```

   Any list ordering of the keys is acceptable (and will probably be determined by Python).

2. Write a Python script that takes a positive amount of dollars as input and prints a number of five dollar bills and one dollar bills that total that amount. It should use as many five dollar bills as possible to express that amount. It should work as shown below:

```
% python3 dollars.py
Enter an amount in dollars: 34
6 fives 4 ones

% python3 dollars.py
Enter an amount in dollars: 30
6 fives

% python3 dollars.py
Enter an amount in dollars: 21
4 fives 1 one

% python3 dollars.py
Enter an amount in dollars: 4
4 ones

% python3 dollars.py
Enter an amount in dollars: 6
1 five 1 one
```

Note that when there are no fives or no ones, those bills aren't reported. When there is only one five dollar bill the word `five` is used. When there is only single one dollar bill the word `one` is used.

Note also that, since the amount input is assumed to be positive, the script should always output some number of five dollar or one dollar bills.

3. 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
```

4. 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.

5. Below is the code for a `Animal` class that could be used in a zoo simulation. (You can imagine that the code for the larger simulation calls the `update` method at every time step.)

```
class Animal:

    def __init__(self, id, species):
        self.id = id
        self.species = species
        self.hunger = 0
        self.energy = 100
        self.asleep = False

    def update(self):
        if self.asleep:
            self.energy += 1
            if self.energy >= 100:
                self.asleep = False
        else:
            self.energy -= 1
            if self.energy <= 0:
                self.asleep = True

    def eat(self, amount):
        if self.asleep:
            self.asleep = False
        self.hunger = max(0,self.hinger - amount)
```

Define a new class `Elephant` to represent elephants. Its species is always `'elephant'` so its constructor doesn't take that information. Unlike many animals, elephant don't wake up to eat, so its `eat` method should do nothing if an elephant is asleep. Otherwise, elphants behave like any other animals.

6. Write a function `occurs_in_consecutives`. It takes a list of consecutive integers from 0 up to some maximum value, where some of the integers are repeated, and also some positive integer less than the maximum value. The function should efficiently determine how many times that value appears in the list. It should run in $O(\log(n))$ time for a list of length n.

   For example:

```
>>> occurs_in_consecutives([0,0,1,2,2,2,2,3,3,4,5,6,6,6,7], 2)
4
>>> occurs_in_consecutives([0,0,1,2,2,2,2,3,3,4,5,6,6,6,7], 4)
1
>>> occurs_in_consecutives([0,0,1,2,2,2,2,3,3,4,5,6,6,6,7], 6)
3
```

7. A computer *bitmap image* is a table of 0s and 1s with a certain number of rows and columns. A bitmap file starts with a line that gives the number of rows and the number of columns. The remaining lines are all the rows in the table. Each row has a number of 0s and 1s equal to the number of columns.

   Write the code for a function called `reverseBitmap`. It takes the name of a bitmap file to read as input, and the name of a file to output. After running the function, it should produce a bitmap file where all the 1s of the input file have become 0s, and all the 0s have become 1s.

   For example, suppose you ran

   ```
   reverseBitMap("bitmap.txt", "reverse.txt")
   ```

   and `bitmap.txt` had these file contents:

   ```
   3 4
   0 1 1 1
   1 0 1 0
   0 1 1 1
   ```

   Then the resulting contents of `reverse.txt` should be the following:

   ```
   3 4
   1 0 0 0
   0 1 0 1
   1 0 0 0
   ```

8. Write the code for a function `webPageAvailable`. It takes a URL for a web page at a site, which is a string (for example `'http://reed.edu/academic-calendar/'`). It also takes a positive `number` of attempts. It can use a module `http`. We assume that module defines a function `fetchURL` that fetches and returns the contents of that page as a (possibly long) string. This function can raise an error `http.TimeoutException` if the web site for the page doesn't respond quickly enough. It can also raise an error `http.NotFound` if the site doesn't exist or the site responds that the page isn't available at that site.

   Your function should return `True` if the web page was fetched successfully in `number` or fewer attempts. It should return `False` otherwise. It should reattempt to fetch the page when a timeout exception occurs and the number of attempts hasn't exceeded `number`.

9. Consider our `BSTree` class. Its structure contains dictionary entries, held as nodes, arranged in order by their keys, and with each key having an associated value. Write a `BSTree` method `sumValues` that computes and returns the sum of the values associated with the keys in the tree.

10. Give the asymptotic running time for each of the following functions. Write this using Theta notation, fully simplified, e.g., $\Theta(n^2)$, $\Theta(n \log(n))$, etc. For each function, the input is a list. Use n to denote the number of items in the list. Note that some of the later functions call the earlier ones.

(a)
```
def sum_alternating(lst):
    i = 0
    total = 0
    while i < len(lst):
        total += lst[i]
        i += 2
    return total
```

(b)
```
def sum_products(lst):
    total = 0
    for x in lst:
        for y in lst:
            total += x*y
    return total
```

(c)
```
def sum_halving(lst):
    i = len(lst) - 1
    total = 0
    while i > 0:
        total += lst[i]
        i = i // 2
    return total
```

(d)
```
def sort_alternate(lst):
    slst = mergesort(lst)
    return sum_alternating(slst)
```