CSC148, Lab #4

This document contains the instructions for lab number 4 in CSC148H. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work.

General rules

We will use the same general rules as for the previous labs (including pair programming).

Overview

In this lab, you will practice reading, and then writing recursive functions.

Recursive functions model the solution to recursive problems: problems that can be broken down into **smaller** instances of a similar problem. Most of the work is done up front — thinking about the solution before you begin writing a lot of code, and thinking of it as made up of similar subproblems. In what follows we call this the **recursion insight**. In most cases we will provide you with the insight, and leave you the task of applying it to either trace or write some code.

Tracing recursion

The next two exercises are paper-and-pencil, and there is neither driver nor navigator. Please discuss with your partner!

Greatest Common Denominator — GCD

This very efficient algorithm is over 2000 years old, and is credited to Euclid (yes **that** Euclid). It is a technique to find the largest non-negative whole number that divides two different numbers, `n1` and `n2`. You could certainly do this by listing all the divisors of `n1` and `n2`, and then finding the biggest number that occurs on both lists, but Euclid made the following labour-saving observation:

Recursion insight: In general, the GCD of `n1` and `n2` is the same as the GCD of `n2` and (`n1 % n2`) — the remainder after dividing `n1` by `n2`.

A special case occurs when `n2` is zero — then `%` doesn't make sense, but the GCD will be `n1` (why?). An extra-special case occurs when both numbers are zero — in that case the GCD is **decreed** to be `0`!

This gives you enough background to trace the following code. (Python already has a built-in **gcd** function in its **fractions** module; we are re-inventing it for purposes of teaching and learning recursion.)

```
def csc_gcd(n1: int, n2: int) -> int:
    """Return the GCD of non-negative integers n1 and n2

    >>> csc_gcd(0, 0)
    0
    >>> csc_gcd(5, 0)
    5
    >>> csc_gcd(15, 35)
    5
    """
    # The gcd of n1 and n2 is n1 if n2 is zero, otherwise it is
    # the same as the gcd of n2 and (n1 % n2)
    if n2 > 0:
        return csc_gcd(n2, n1 % n2)
    else:
        return n1
```

Now, trace the following calls **in order**. It is important to plug in a value when you see a recursive call you have already solved, **rather than tracing any further**! We've filled in the first example of what we mean by "tracing."

1. Trace **csc_gcd(5, 0)**

   ```
   csc_gcd(5, 0)  --> 5
   ```

2. Trace **csc_gcd(15, 5)**. Be sure to plug in the value of **csc_gcd(5,0)** directly **without further tracing**

3. Trace **csc_gcd(35, 15)**. Be sure to plug in the value of **csc_gcd(15, 5)** directly **without further tracing**

After you've discussed with your partner, call your TA over and show your work.

Binary representation

In the base 2 number system there are only two possible digits: 0 and 1. These are often called bits (for **bi**nary dig**its**). Each bit is an increasing power of 2 (that is, each bit counts for twice as much as the bit to the right), so the binary number 101 represents the value:

$$(1 \times 4) + (0 \times 2) + (1 \times 1) = 5$$

Here is a table of the first 9 non-negative numbers in both decimal and binary.

Decimal	Binary string
0	"0"
1	"1"
2	"10"
3	"11"
4	"100"
5	"101"
6	"110"
7	"111"
8	"1000"

Recursion insight: In binary, multiplying by two is the same as adding a zero on the right-hand side (sometimes called a left-shift). Dividing by two is the same as removing the right-most bit (sometimes called a right-shift). Taken together, these mean that:

1. The right-most bit of a binary number is "0" for all even binary numbers (two times something), and "1" for all odd binary numbers.
2. The binary string that precedes the right-most bit of a binary number is just the binary string representing that number divided by two — integer-divided, or `n // 2`.

Those two ideas give you the tools to follow this definition, but it will certainly help to trace it (below):

```
def bin_rep(n: int) -> str:
    """Return the binary representation of n

    >>> bin_rep(0)
    "0"
    >>> bin_rep(1)
    "1"
    >>> bin_rep(5)
    "101"
    """
    # The binary representation of n is the binary representation of
    # n // 2 concatenated with the binary representation of n's last bit
    # provided n > 1, otherwise it's just n as a string
    if n > 1:
        return bin_rep(n // 2) + bin_rep(n % 2)
    else:
        return str(n)
```

Now trace the following calls **in order**. Again, if you see a recursive call that you have already traced, fill in its value directly **without further tracing**.

1. Trace `bin_rep(0)`

```
bin_rep(0)  -->  "0"
```

2. Trace `bin_rep(1)`

3. Trace `bin_rep(2)`. Be sure to plug in the value of any recursive calls to **bin_rep(0)** or **bin_rep(1)** directly **without further tracing**

4. Trace `bin_rep(5)`. Be sure to plug in the value of any recursive calls to **bin_rep(2)** directly **without any further tracing**

Once you've finished arguing with your partner, call over your TA and show them your results.

Now, you write the recursive code

In all the exercises below, you should begin as follows:

- Write the function header (including meaningful function name!) and docstring with some examples:

```
def f(n: int) ->:
    """
    Single-sentence describing f

    >>> f(3)
    expected return of f(3)
    ...
    """
```

Structured lists

(Student `s1` drives, student `s2` navigates)

Start Idle in a new directory called `lab04` and start a new file called `recursive_examples.py`. Before you start solving the problem below, type:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

. . . at the end of of your file, so that it will later automatically test all the docstring examples when you run your code. This is lightweight unit testing.

You have likely used the Python operator "`in`" to determine whether a specific element exists in a list. For example:

```
>>> 3 in [1, 2, 3, 5]
True
>>> 3 in [1, 2, 5]
False
```

You probably have enough intuition now to search a nested list to determine whether it contains a specific element. However, such a search would likely take work that is proportional to the size of the list — if the element is not in the list, you would never really know you were done until you had examined each list element.

Consider the advantage of particularly structured nested lists, which we'll call `SearchLists`. These are defined:

SearchList: is either `None` or a python list of 3 elements

(0) Element 0 is an integer

(1) Element 1 is either `None` or a `SearchList` containing integers smaller than Element 0

(2) Element 2 is either `None` or a `SearchList` containing integers larger than Element 0

Here's a small example of a `SearchList`

```
[5, [2, [1, None, None], [3, None, None]], [6, None, None]]
```

Use the properties of a `SearchList` to figure out how to determine whether a given integer is somewhere in the `SearchList`. Notice that if a given integer is less than element 0, it is either in element 1 somewhere, or not in the `SearchList` at all. Similarly, if a given integer is greater than element 0, it is either in element 2 somewhere, or not in the `SearchList` at all.

Recursion insight: The task of searching for a number in either of the "child" `SearchLists` at element 1 or element 2 is the same as the task of searching the entire `SearchList`.

Use these ideas to complete the implementation of `find_num` below. Don't forget to start by writing some examples in the docstring!

```
def find_num(SL: 'SearchList', n: int) -> bool:
    """
    Return True if n is in SL, False otherwise

    >>> ... some examples, please!
    ...
    ...
    """
```

If you get stuck, call your TA over for some hints.

If you're not stuck, trace through simple examples — un-nested lists — to convince yourself your code works. Then trace through the next most-complicated example — a list containing at least one un-nested list — to convince yourself that your code works. Remember to treat the simple cases you've already traced as black boxes, or already solved problems. Then show your TA your work.

Freezing list copies

(Student `s2` drives, student `s1` navigates)

Usually, when you copy a Python list, you store a reference to it. This means that if the original list changes, your reference leads you to the changed list. Often you want this behaviour.

Sometimes, however, you'd like a copy of all the values of a list where all its elements (and elements of elements, if the list contains sub-lists) remain as they were at the moment you copied them, not subject to changes that might be invoked in some other part of your code, or even somebody else's code. Python provides a deep copy function, `copy.deepcopy`, for that.

Here's an idea. It's not as complete, or difficult, as `copy.deepcopy` (for example, it won't deal with lists that contain references to themselves). But it's a very powerful first cut.

Recursion insight: In order to `freeze` a list, you must `freeze` any of its elements that are also lists. In other words, as you create the new, frozen, version of a list,

1. If the corresponding old element is a non-list, simply make a reference to it (this is what normally happens when we assign expressions in Python)
2. If the corresponding old element is a list, treat it just as you do its enclosing list — produce a new list by assigning new elements as elements as in these two steps (think recursion!)

Use your ideas to complete the implementation of the function `freeze` below. Here's a small example of how it should work:

```
>>> L1 = [1, [2, 3], 4]
>>> L2 = freeze(L1)
>>> L1 is L2
False
>>> L1[1] is L2[1]
False
>>> L1 == L2
True
>>> L1[1] == L2[1]
True
```

```
def freeze(X: object) -> object:
    """
    If X is a list, return a new list with equivalent contents,
    and recursively treat the contents of X as you treated X itself...
    If X is not a list, return X itself

    >>> ... don't forget examples!
    ...
    ...
    """
```

If you're stuck, call your TA over and show your work.

If you're not stuck, trace through simple examples — un-nested lists — to convince yourself your code works. Then trace through the next most-complicated example — a list containing at least one un-nested list — to convince yourself that your code works. Remember to treat the simple cases you've already traced as black boxes, or already solved problems.

Be sure to show your TA your work.

Extra, optional problems

If you finish with the problems above, here are some other ones to try. These are certainly not required for completing the lab.

- Write a recursive function `rev_string(s)` that returns the reversal of string `s`.

- Extend `bin_rep(n)` to create `base_rep(n,k)` to give a string representing non-negative integer `n` in base `k`, where $2 \leq k < 10$. In base `k`, the only digits allowed are $\{0, \ldots, k-1\}$.
- Extend `freeze(X)` so that it also handles tuples and dictionaries.
- For a list of distinct integers, `L`, define `switches(L)` as the number of pairs in `L` that are not in increasing order. For example, `switches([3,1,2])` returns 2, since (3, 1) and (3,2) are out of order. Implement the function `switches(L)` recursively.