Computational Biology
Tufts University Bio 35
Fall 2021

Assignment 5: Recursion & memoization

Recursion: All of your code for this problem should go in a file called `Recursion.py`. This assignment is taken (with minor adaptations) from the companion material for CFB. You can find the original at `https://www.cs.hmc.edu/twiki/bin/view/CFB/RecursionWarmup`.

Introduction

This assignment is the only one that has relatively weak biological motivation. Sorry! Recursion is a fundamental computational tool that we will need for the next section. Write each of the following functions using recursion. (Do not use for loops or while loops for these problems.)

1. `power(x, y)` takes two non-negative integers `x` and `y` as input and returns the value `xy`. Of course, we could do this simply using the notation `x**y`, but the point here is to do this using recursion and multiplication. Note that `x**0` is 1 by definition.

example
```
>>> power(2, 3)
8
>>> power(10, 2)
100
```

2. `dot(L, K)` should output the dot product of the lists `L` and `K`. The dot product of two lists is the sum of the products of the elements in the same position in the two lists. You may assume that the two lists are of equal length. If these two lists are both empty, `dot` should output 0.0. Assume that the input lists contain only numeric values.

example
```
>>> dot([5,3], [6,4])  # <-- Note that 5*6 + 3*4 = 42
42
```

3. `GCcount(DNA)` takes DNA string as input and returns the number of G's and C's that appear in that string.

example
```
>>> GCcount('TGTCG')
3
>>> GCcount('ATATAT')
0
```

4. `countStarts(DNA)` takes a DNA string as input and returns the number of times the string `'ATG'` appears in that string. We did this problem in Chapter 3 using for loops. The objective here is to do it using recursion!

example
```
>>> dot([5,3], [6,4])   <-- Note that 5*6 + 3*4 = 42
42
```

5. `explode(S)` takes a string S as input and should return a list of the characters (each of which is a string of length 1) in that string. For example:

example
```
>>> explode('spam')
['s', 'p', 'a', 'm']

>>> explode ('')
[]
```

Note that Python is happy to use either single quotes or double quotes to delimit strings - they are interchangeable but if you use a single quote at the start of the string you must use one at the end of the string (and similarly for double quotes). For example:

example
```
>>> 'spam' == "spam"
True
```

6. `ind(e, L)` takes in an element `e` and a sequence `L` where by “sequence” we mean either a list or a string; fortunately indexing and slicing works the same for both lists and strings, so your ind function should be able to handle both types of input! Then, ind should return the index at which e is first found in `L`. Counting begins at 0, as is usual with lists. If `e` is NOT an element of `L`, then `ind(e, L)` should return an integer that is at least the length of `L` (that’s a signal that the user will interpret as “the element that I’m looking for isn’t in the list or string”).

example
```
>>> ind(42, [ 55, 77, 42, 12, 42, 100 ])
2
>>> ind(42, range(0,100))
42
>>> ind('hi', [ 'hello', 42, True ])
3
>>> ind('hi', [ 'well', 'hi', 'there' ])
1
>>> ind('i', 'team')
4
>>> ind('p', 'spam')
1
```

7. `removeAll(e, L)` takes in an element `e` and a list `L`. Then, `removeAll` should return another list that is identical to `L` except that all elements identical to `e` have been removed.

example
```
>>> removeAll(42, [ 55, 77, 42, 11, 42, 88 ])
[ 55, 77, 11, 88 ]

>>> removeAll(42, [67, "I", "love", "spam", 100])
[67, "I", "love", "spam", 100]  # <-- notice that 42 wasn't in this list,
                                # so we got an identical list back!
```

8. **Challenge problem**: Deep Cleaning with Recursion! This problem demonstrates something that truly requires recursion! The `removeAll` function above does this:

example
```
>>> removeAll(42, [42, 67, 42, [42, 42, 43], 47])
[67, [42, 42, 43], 47]
```

In other words, in this example, it removes all of the 42's from the list, but it does not remove 42's that are embedded in lists within that list. Your job here is to write a function called `deepRemoveAll` that scours its input and removes not only the desired item, but also removes that item deep inside other lists. Here are a few examples:

example
```
>>> deepRemoveAll(42, [42, 67, 42, [41, 42, 43], 47])
[67, [41, 43], 47]
>>> deepRemoveAll(47, [42, 47, [1, 2, [47, 48, 49], 50, 47, 51], 52])
[42, [1, 2, [48, 49], 50, 51], 52]
```

You'll need one ingredient that we haven't seen yet, namely, the ability to test whether something is a list or not. If you have an item `X` (we chose that name just for example; the name of the item could be anything!) and you wish to test whether it's a list, you can use the Boolean:

example
```
type(X) == type([])
```

This returns `True` if item `X` is a list (it's the same "type" as the empty list - we could have used any other list on the right-hand side for this comparison), and returns `False` otherwise.

9. Imagine that you're invited to enter a vault and help yourself to as much as you can pack into your knapsack. Your knapsack, of course, has limited weight capacity. Each item in the vault is marked with its weight and value. Your objective is to select a subset of the items in the vault such that their total weight does not exceed your knapsack's capacity and such that the total value of the items that you choose is maximized. Each item can only be used once.

   Here is an example of us defining some items in `Python`:

example
```
>>> rock = [160, 4]                 # Heavy and low value
>>> diamond = [3, 25]               # Light and very valuable!
>>> spam = [2, 1]                   # Very light weight and some value
>>> stuff = [rock, diamond, spam]   # Now make a list of those items
>>> stuff
[ [160, 4], [3, 25], [2, 1] ]
```

   Of course, we could have just defined stuff directly as

example
```
stuff = [ [160, 4], [3, 25], [2, 1] ]
```

   Your job is to write a function `knapsack(capacity, items)` that takes a positive integer capacity and a list of items (such as stuff in the example above) where each item in the list is itself a list of the form `[weight, value]`. Then knapsack returns the best solution, which is maximum total value of items you can pack. Below is an example:

example
```
>>> knapsack(5, stuff)    #  <-- knapsack capacity is 5
26                        #  <-- The optimal value is 26 (uses the diamond and spam)
```

   Of course, we could have also run this without first defining stuff as in:

example
```
>>> knapsack(5, [ [160, 4], [3, 25], [2, 1] ])
26
```

10. In Chapter 6 we wrote the subset function that determines whether or not there exists a subset of the `numberList` that adds up to the target. The code that we developed is here:

example
```
def subset(target, numberList):
    ''' Returns True if there exists a subset of numberList that adds
        up to target and returns False otherwise.'''
    if target == 0: return True
    elif numberList == []: return False
    if numberList[0] > target: return subset(target, numberList[1:])
    else:
        useIt = subset(target - numberList[0], numberList[1:])
        loseIt = subset(target, numberList[1:])
        return useIt or loseIt
```

Unfortunately, this function can become very slow as the inputs become "large". Cut-and-paste into your own file and try running it on the following

example
```
>>> subset(1234567, range(2, 100, 2))
```

(What is that `range(2, 100, 2)` doing?)

After figuring that out, you might want to go out for coffee with your friends, and perhaps a movie, before checking if your program has completed running. Alternatively, you can press `CTRL-C` to stop your program. The `subset` function is slow because it does a large number of redundant calculations. Your task now is to write a much faster memoized version called `memoizedSubset`. As we saw with the change function that we memoized in the text, we will want to give this new function a tuple input instead of a list. Our new function will look like this: `memoizedSubset( target, numberTuple, memo )`.

After you've written it, try the same inputs with this new, faster function:

example
```
>>> numberTuple = tuple(range(2, 100, 2))
>>> memoizedSubset(1234567, numberTuple, {} )
```