Some Recursion Practice Problems

Jon Fast
jonathan.fast@msu.montana.edu

April 13, 2014

1 Intro

I've taken the liberty of putting some interesting (intermediate to challenging) recursion problems together to help all of you in Computer Coding practice solving some more challenging problems. Try them out on your own (with the exception of #s 4 and 5 given their difficulty levels, you probably can try figuring out how they work anyways). Remember, practice makes perfect!

2 Problem 1: Decimal To Binary Numbers

2.1 Problem Definition

The idea of this problem is to convert an integer to a binary string. A binary string is a string of numbers consisting of only 0s and 1s. This string is the binary representation of a decimal number. The problem is solved by dividing a number by two and returning its remainder as a 0 or a 1. The problem looks something like this recursively: ((((number // 2) // 2) // 2) // 2)...etc. until number = 0. Upon returning the empty string when the number passed to the function is zero, we merely append the remainder of the current division operation to the string (which is only 0 or 1, given that we're dividing by two) until we return the entire binary string at the end of the recursion.

2.2 Solution

The code of interest:

```
# decimal to binary (easy)
def dec_to_bin_string(n):
    # if division by two is zero
    if n == 0:
        # return the empty string
        return ''
    else:
        # otherwise, find the result of dividing the
        # current number by two, and append the remainder
        # of that division to the current binary string
        return dec_to_bin_string(n // 2) + str(n % 2)

# test it!
print("19 to binary string:", dec_to_bin_string(19))
```

Yields some sample output:

```
*
19 to binary string: 10011
```

3 Problem 2: The Pell Numbers

3.1 Problem Definition

The idea of this problem is similar to the fibonacci numbers. If a number passed recursively to a function is 0, then return 0, and if it's 1, then return one. Otherwise, return 2 multiplied by the current Pell number - 1 + the current Pell number - 2 or:

$$pellNumber_{current} = (2 \times pellNumber_{n-1}) + pellNumber_{n-2}$$

which looks an awful lot like:

$$fibonacciNumber_{current} = fibonacciNumber_{n-1} + fibonacciNumber_{n-2}$$

with the addition of the $(2 \times whatever)$ portion of the equation. Just like the Fibonacci numbers, the Pell numbers are self-referencing.

3.2 Solution

The code of interest:

```
# pell numbers (moderate)
def pell_number(n):
    # 0 when n = 0
    if n == 0:
        return 0
    # 1 when n is 1
    elif n == 1:
        return 1
    else:
        # otherwise, 2 times the last pell number
        # plus the pell number before it
        return 2 * pell_number(n - 1) + pell_number(n - 2)

# test it (pell numbers are available online for reference)
print("12th Pell Number:", pell_number(12))
```

Yields some sample output:

*

12th Pell Number: 13860

4 Problem 3: Finding the First Position of a Given Character in a String

4.1 Problem Definition

The idea of this problem is to go through a given string until we find the first character in that string matching a character passed into the function. The idea is to start at position 0 in the list, and go to the next position until we hit the length of the list, or until we find a character that matches one we've passed into the function. The three things that must be passed into this function are:

1. The first position in the list (or 0)
2. The list we want to iterate over
3. The character we want to find.

When we want to go to the next position in the list, the next function call gets $position_i + 1$. The character we are searching for should stay the same throughout each next recursive function call. The list we want to iterate over should also stay the same. The base cases for this function are:

1. Once the position value reaches the end of our list, we should return no value.

2. If the value in our current position on the list is equivalent to the one given, we return its position.

Otherwise, we iterate through the list recursively by calling our function on the next position in the list.

4.2 Solution

The code of interest:

```
# finding the first position of a
# given character in a string (slightly more challenging)
def find_character(lst, pos = 0, char='A'):
    # we shouldn't iterate over a negative
    # position in the list
    if pos < 0:
        pos = 0
    # once we've hit the end of the list
    # we're done, and if we haven't found
    # a character, none is returned
    if pos == len(lst):
        return None
    # if we hit the characted in the list
    # we've been searching for, the search
    # is over, and we backtrack
    if lst[pos] is char:
        return pos
    else:
        # otherwise, we continue scattering and looking
        return find_character(lst, pos + 1, char)

# test it!
print("Position: ", find_character(['1', '3', '2', '7', '5', '6'], 0, 's'))
print("Position: ", find_character("1234586strange7string", 0, '7'))
```

Yields some sample output:

```
    *
Position: None
```

Position: 14

5 Problem 4: String Reversal

5.1 Problem Definition

The idea of the problem is to reverse a string using recursion. Imagine the problem as simply flipping the last and first character of a smaller and smaller string. One we flip the characters at the end of the string, we 'push' the indices of the string inwards (increase the first index, decrement the last) so that we're solving the reversal problem on a smaller string. Once the last index is less than or equal to the first, then we return the produced string.

5.2 Solution

The code of interest:

```
# imports
import sys

# reversing a string (challenging)
def reverse(string, firstpos = 0, lastpos = sys.maxsize):
    # fix string position to zero at start
    if firstpos < 0:
        firstpos = 0
    if lastpos >= len(string):
        lastpos = len(string) - 1
    if lastpos <= firstpos:
        return string
    else:
        # perform reversal for one character
        newstring = string[:firstpos] + \
        string[lastpos] + \
        string[firstpos + 1:lastpos] + \
        string[firstpos] + \
        string[lastpos + 1:]
        # return the string with the next character reversed
        return reverse(newstring, firstpos + 1, lastpos - 1)
```

```
# test it!
stringList = ['balloon', 'baboon', \
              'aardvark', 'computer', \
              'the quick brown fox jumps over the lazy dog']
for i in range(len(stringList)):
    print("Reverse of", stringList[i], "is:", reverse(stringList[i]))
```

Yields the sample output:

*

Reverse of balloon is: noollab
Reverse of baboon is: noobab
Reverse of aardvark is: kravdraa
Reverse of computer is: retupmoc
Reverse of the quick brown fox jumps over the lazy dog is: god yzal eht revo spmuj xof nworb kciuq eht

6 Problem 5: Counting the Number of Words in a String (aka. . . String Tokenizing)

6.1 Problem Definition

The idea of the problem is to go through the string and determine how many words are in it. Be aware that this problem is very hard, and the code given will only work if there is one space between each word. This is the only problem I don't expect you to know how to do, but it is a useful study if you'd ever want to build a tokenizer for a compiler.

6.2 Solution

The code of interest:

```
# finding the number of words in a string,
# and returning each word found
# note: this method will fail if there is are multiple spaces
# in between words or at the beginning or end of the string
# notice that we can list 'found' words intermediately
# (very challenging)
def number_of_words(string, pos = 0, count = 0, temp = ""):
    if pos < 0:
```

```
        pos = 0
    if not string:
        return 0
    if pos == len(string) and not temp:
        return count
    elif pos == len(string) and temp:
        print("Found:", temp)
        temp = ""
        return count + 1
    else:
        if string[pos] == ' ':
            count += 1
            print("Found:", temp)
            temp = ""
            return number_of_words(string, pos + 1, count, temp)
        else:
            temp += string[pos]
            return number_of_words(string, pos + 1, count, temp)

# test it!
sentence = "Number of words in this sentence"
print(sentence + ":", number_of_words(sentence))
sentence += " plus two"
print(sentence + ":", number_of_words(sentence))
```

Yields the sample output:

*

Found: Number
Found: of
Found: words
Found: in
Found: this
Found: sentence
Number of words in this sentence: 6
Found: Number
Found: of
Found: words
Found: in
Found: this

Found: sentence

7 Fin!

Remember to practice, practice, practice for the practicum (which isn't too far ahead!)