Lab 13 – Off to See the Wizard

Keep the following guidelines in mind when writing recursive programs:

- Base Case:
  solve the simplest version (or versions) of the problem directly
- General Case:
  (a) make the problem slightly smaller
  (b) let the wizard (*i.e.*, the recursion) "magically" solve the smaller problem for you
  (c) use the result of (b) to help you solve the original problem

Download and unzip **lab13_files.zip** from the class web page, and open **lab13.py.** This file is set up to use the autotester program. For example, to test your `addup` function, just type `test(addup)`.

1. Write a recursive function called **addup(*nums*)** that takes a list of numbers and <u>returns</u> their sum. A list containing no numbers should sum to 0. For a non-empty list, let the recursion add up most of the numbers for you (all of them except the first). You are not allowed to use a loop. For example:

```
>>> addup([5, 4, 3, 2, 1])
15
>>> addup([6, 5, 4, 3, 2, 1])
21
```

2. Write a recursive function called **smallest(*nums*)** that takes a list of numbers, which you may assume will not be empty, and <u>returns</u> the smallest one in the list. If the list has only one number, the answer is obvious. Otherwise, let the recursion figure out the smallest number appearing *after* the first number in the list, then compare that to the first number. You are not allowed to use Python's `min` function. For example:

```
>>> smallest([5, 4, 3, 2, 1])
1
>>> smallest([2, 7, 5, 9, 3, 9])
2
>>> smallest([9])
9
```

3. Write a recursive function called **countfives(*nums*)** that takes a list of numbers and <u>returns</u> how many fives appear in the list. An empty list contains no numbers, so the result should just be 0. For a non-empty list, let the recursion figure out how many fives appear *after* the first number in the list, then test the first number directly to see if you need to modify the recursion's answer. Example:

```
>>> countfives([5, 10, 5, 5, 20, 30, 5])
4
>>> countfives([10, 20, 30, 5, 40])
1
```

4. Write a recursive program called **reverse(*s*)** that takes a string *s* and <u>returns</u> its reversal. The reversal of the empty string is just the empty string. For a non-empty string, remove the first letter from the string, and let the recursion reverse the remaining portion of the string for you. Then attach the first letter to the end.

```
>>> reverse('')
''
>>> reverse('watermelon')
'nolemretaw'
```

5. Write a recursive function called **double(*s*)** that takes a string as input and <u>returns</u> a new string with all of the original letters doubled. Doubling an empty string just gives the empty string. For a non-empty string, let the recursion double most of the string for you (all letters except the first). For example:

```
>>> double('')
''
>>> double('watermelon')
'wwaatteerrmmeelloonn'
```

6. Write a recursive function called **swap(*letter1, letter2, s*)** that takes two letters and a string and returns a new version of the string with all occurrences of *letter1* and *letter2* swapped. Hint: let the recursion do the swapping in the part of the string beyond the first letter, and use if-tests to decide which letter to attach to the front of the new string.

```
>>> swap('o', 'i', 'lollipop')
'lillopip'
>>> swap('l', 'p', 'lollipop')
'poppilol'
>>> swap('o', 'x', 'lollipop')
'lxllipxp'
```

7. Write a recursive function called **commas(*digits*)** that takes a string of digits such as "1234567" as input and returns a new digit string with commas inserted in all the appropriate places. If the input string contains three or fewer digits, inserting commas is not necessary. For longer strings, insert one new comma at position -3 (third from the right), and then let the recursion insert all the other commas to the left of this position for you.

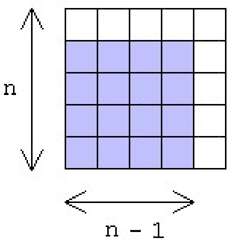
```
>>> commas('123')
'123'
>>> commas('1234')
'1,234'
>>> commas('1234567')
'1,234,567'
>>> commas('123456789000')
'123,456,789,000'
```

8. Write a recursive function called **fib(*n*)** that returns the *n*th Fibonacci number. The *n*th Fibonacci number is just the sum of the previous two Fibonacci numbers in the sequence. By definition, the first two Fibonacci numbers, fib(1) and fib(2), are both 1. The beginning of the sequence is shown below:

1 1 2 3 5 8 13 21 34 55 89 . . .

```
>>> fib(1)
1
>>> fib(10)
55
```

9. Write a recursive function called **square(*n*)**, which computes n^2 using only addition and subtraction. You are not allowed to use multiplication or exponentiation. Hint: consider the picture below, in which square(n) is equal to the number of boxes in the grid. To reduce the problem to a smaller size, consider just the shaded area, which represents the value of $(n-1)^2$.



10. Write a recursive function called **subsets(*letters*)** that takes a string of letters with no duplicates as input such as `'abc'`, representing a set of letters, and returns a *list* of strings representing all the unique subsets that can be made from the letters. The order of the letters in a string does not matter (*e.g.,* `'abc'` and `'bac'` would represent the same subset). The order in which the strings appear in the final list does not matter, either. The empty string has exactly *one* possible subset, namely the empty string itself. (Hint: to construct the final answer from the result of the recursion, it will be helpful to use a for-loop.) Examples:

```
>>> subsets('bc')
['bc', 'b', 'c', '']
>>> subsets('abc')
['abc', 'ab', 'ac', 'a', 'bc', 'b', 'c', '']
>>> subsets('')
['']
```

11. Write a recursive function called **anyeven(*nums*)** that takes a list of numbers and returns True if any number in the list is even, or False otherwise. What is an appropriate answer for the empty list? Hint: a number `n` is even if `n % 2 == 0`. Examples:

```
>>> anyeven([3, 5, 6, 7, 9, 10])
True
>>> anyeven([2, 3, 5, 7, 9])
True
>>> anyeven([3, 5, 7, 9, 11])
False
```

12. Write a recursive program called **power(*b*, *n*)** that computes and returns the value b^n, where n is any integer exponent ≥ 0 and b is any base. By definition, b^0 equals 1 for any b. In general, b^n equals b^{n-1} times b. For example, to compute 2^4, all your program needs to do is compute 2^3 and then multiply that value by 2. To compute 10^5, it can just compute 10^4 and multiply that value by 10. Examples:

```
>>> power(2, 4)
16
>>> power(5, 0)
1
>>> power(10, 5)
100000
```

13. The *Fibonacci sequence* of numbers described in Exercise 8 above was defined by the famous Italian mathematician Fibonacci around the year 1200. Its recursive definition is given below:

$\text{Fib}(1) = 1$
$\text{Fib}(2) = 1$
$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

Much later, the French mathematician Lucas defined his own sequence of numbers 2, 1, 3, 4, 7, 11, 18, 29, 47, … , nowadays called the *Lucas sequence*, using a similar recursive rule, shown below:

$\text{Lucas}(1) = 2$
$\text{Lucas}(2) = 1$
$\text{Lucas}(n) = \text{Lucas}(n-1) + \text{Lucas}(n-2)$

Not to be outdone by Fibonacci or Lucas, the not-so-famous American computer scientist Marshall very recently defined *his* own sequence of numbers with a much more impressive-looking recursive definition:

$\text{Marshall}(1) = 2$
$\text{Marshall}(2) = 2$

$$\text{Marshall}(n) = \frac{(\text{Marshall}(n-1))^2 + (\text{Marshall}(n-2))^3}{3 \times \text{Marshall}(n-2)}$$

Unfortunately, for some reason this sequence never really caught on, despite its obviously much more sophisticated mathematical structure (compared to Fibonacci's and Lucas's simple formulas). Write a recursive function called **marshall(*n*)** that computes and returns the nth Marshall number. What are the first ten Marshall numbers? Can your code be simplified? If so, define a simpler version of your function called **marshall2(*n*)** and explain in a comment in your code why it is equivalent to the original.

14. Since the decimal (base 10) number 970 ends in 0, it equals 10 times 97. Likewise, since the binary (base 2) number 101010 ends in 0, it equals 2 times the value of the binary number 10101. Similarly, 971 equals 10 times 97, plus 1, and 101011 equals 2 times the value of 10101, plus 1. Based on this idea, write a recursive function called **binary2decimal(*binstring*)** that takes a non-empty string of binary digits such as "101010" and returns its decimal equivalent. Examples:

```
binary2decimal('101010')  =>  42
binary2decimal('10101')   =>  21
binary2decimal('101011')  =>  43
binary2decimal('1111')    =>  15
```