Eric Roberts
CSCI 121

Handout #33
October 10, 2018

Assignment #5—Recursion and Lists

Due: Wednesday, October 24, 11:59 P.M.

Problem 1

Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Write a recursive function `cannonball` that takes as its argument the height of the pyramid and returns the number of cannonballs it contains. Your function must operate recursively and must not use any iterative constructs, such as `while` or `for`.

Problem 2

As you know from Chapter 4, the mathematical combinations function $C(n, k)$ is usually defined in terms of factorials, as follows:

$$C(n, k) \;=\; \frac{n!}{k! \times (n-k)!}$$

The values of $C(n, k)$ can also be arranged geometrically to form a triangle in which n increases as you move down the triangle and k increases as you move from left to right. The resulting structure, which is called ***Pascal's Triangle*** after the French mathematician Blaise Pascal, is arranged like this:

$C(0, 0)$
$C(1, 0) \quad C(1, 1)$
$C(2, 0) \quad C(2, 1) \quad C(2, 2)$
$C(3, 0) \quad C(3, 1) \quad C(3, 2) \quad C(3, 3)$
$C(4, 0) \quad C(4, 1) \quad C(4, 2) \quad C(4, 3) \quad C(4, 4)$

Pascal's Triangle has the interesting property that every entry is the sum of the two entries above it, except along the left and right edges, where the values are always 1. Consider, for example, the circled entry in the following display of Pascal's Triangle:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 ⑮ 20 15 6 1
1 7 21 35 35 21 7 1

This entry, which corresponds to $C(6, 2)$, is the sum of the entries—5 and 10—that appear above it to either side. Use this relationship between entries in Pascal's Triangle to write a recursive implementation of the `combinations(n, k)` function that uses no loops, no multiplication, and no calls to `fact`.

Problem 3

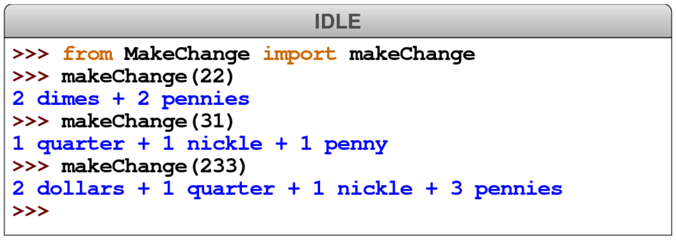
In today's class, I defined the following constant arrays as examples of list initialization in Python:

```
COIN_VALUES = [ 1, 5, 10, 25, 50, 100 ];
COIN_NAMES = [
    "penny",
    "nickle",
    "dime",
    "quarter",
    "half-dollar",
    "dollar"
];
```

Use these definitions to write a function `makeChange(change)` that displays the number of coins of each type necessary to produce `change` cents. You may assume that the currency is designed—as American coinage is—so that the following strategy always produces the correct result:

- Start with the last element in the array (in this case, dollars) and give as many of those as are possible.
- Move on to the previous element and give as many of those as possible, continuing this process until you reach the number of cents.

A sample IDLE session with this function looks like this:



```
>>> from MakeChange import makeChange
>>> makeChange(22)
2 dimes + 2 pennies
>>> makeChange(31)
1 quarter + 1 nickle + 1 penny
>>> makeChange(233)
2 dollars + 1 quarter + 1 nickle + 3 pennies
>>>
```

To duplicate this output, you will also need to implement `createRegularPlural`, which is described in exercise 9 from Chapter 6.

Problem 4

Write a predicate function `isSorted(list)` that takes a list and returns `True` if the list is sorted in increasing order and `False` if there are any elements out of order. Calling `isSorted` on an empty list or a list with one element should always return `True`.