CMSC110 Introduction to Computing

Lab #13: Recursion
Week of November 28, 2016

This lab will give you some practice with a coding technique called *recursion*. A *recursive* function is one that calls itself, or *recurs*. Any repetitive task can be phrased as recursion, and many tasks are more naturally phrased recursively than *iteratively*. (An *iterative* loop is one written with `for` or `while`.)

As an example, let's look at a function to find the maximum element in an array. First, here is the iterative version:

```
float findMax(float[] arr)
{
      float max = arr[0];
      for(int i = 0; i < arr.length; i++)
      {
            if(arr[i] > max)
            {
                  max = arr[i];
            }
      }
      return max;
}
```

For this function to work, it needs to store the running maximum in a variable `max` and then use an index `i` to look through the array. We can simplify this operation by thinking of a mathematical definition of the `findMax` operation. Before we can do so, however, we need to rephrase the question slightly: we will ask for the maximum element that occurs between a certain index and the end of the list. So `findMax(arr, 3)` finds the maximum element with an index that is greater than or equal to 3, and `findMax(arr, 0)` finds the maximum element overall (because all indices are greater than or equal to 0). Now, we can state `findMax` as a mathematical *recurrence relation*:

$$\texttt{findMax(arr, n)} = \begin{cases} \texttt{arr[n]} & \texttt{n == arr.length - 1} \\ \texttt{arr[n]} & \texttt{arr[n] > findMax(arr, n+1)} \\ \texttt{findMax(arr, n+1)} & \text{otherwise} \end{cases}$$

What this means is that the result of running `findMax(arr, n)` is one of three possibilities:

1. If we have only one element of the array left with an index greater than or equal to `n`, then the maximum must be that element. (We know this is the case when `n == arr.length - 1`.)
2. Otherwise, if the current element (that is, `arr[n]`) is greater than the maximum of the rest of the list (that is, `findMax(arr, n+1)`), then return the current element.
3. Otherwise, the maximum is just the maximum of the rest of the list.

We can code this up as the following:

```
float findMax(float[] arr, int n)
{
        if(n == arr.length - 1)
        {
                return arr[n];
        }

        float m = findMax(arr, n);
        if(arr[n] > m)
        {
                return arr[n];
        }
        else
        {
                return m;
        }
}
```

See how the mathematical definition of `findMax` is directly encoded into the Processing code.

Task 1: Put this recursive definition of `findMax` into a fresh Processing sketch.

Task 2: Write a `setup()` function that tests your `findMax`. This `setup()` function must create an array of `float`s, run `findMax` on it, and then print out the result. Verify that the result is as expected.

Task 3: Using `findMax` as a template, write a `findMin` recursive function that returns the minimum element in an array.

If we have an array of one-digit numbers, we can create the number that these digits would form if concatenated together, backwards. That is, we want a function `int build(int[] digits, int n)` that works as follows, where I am using, for example, `{1, 2, 3}` to represent an array with three elements in it, 1, 2, and 3:

- `build({1, 2, 3}, 0)` → `321`
- `build({1, 2, 3}, 1)` → `32`
- `build({5, 0, 4, 1}, 0)` → `1405`
- `build({5, 0, 4, 1}, 3)` → `1`
- `build({5, 0, 4, 1}, 4)` → `0`

See how the `build` function simply concatenates the digits in the array to form a new number. The second parameter, `n`, works as above, saying where in the array we should start looking. Here is a mathematical specification of `build`:

$$\texttt{build(digits, n)} = \begin{cases} \texttt{0} & \texttt{n == digits.length} \\ \texttt{build(digits, n+1) * 10 + digits[n]} & \text{otherwise} \end{cases}$$

Task 4: Write the `build` function in your Processing sketch according to this specification. Try to understand why it works!

Task 5: Test your `build` function by calling it and printing the results from `setup()`.

Strings work quite well in recursive functions. This is because you can easily break apart a string using the `substring` member function. As the Processing reference says:

Description Returns a new string that is a part of the original string. When using the **endIndex** parameter, the string between **beginIndex** and **endIndex**-1 is returned.

Syntax

```
str.substring(beginIndex)
str.substring(beginIndex, endIndex)
```

Parameters		
	str	String: any variable of type String
	beginIndex	int: position from which to begin (inclusive)
	endIndex	int: position from which to end (exclusive)

Let's say that `str` contains the string `"turkey!"`. Then, `str.substring(0,1)` will be the string `"t"` while `str.substring(1)` will be the string `"urkey!"`. Note that if we leave off the second parameter, `substring` gives us every character until the end of the original string.

We want to write a function `countXs` that counts the number of occurrences of the character `x` in a string. Here is the mathematical specification:

$$\texttt{countXs(str)} = \begin{cases} \texttt{0} & \texttt{str}\text{ is empty} \\ \texttt{1 + countXs(str.substring(1))} & \text{the first char in }\texttt{str}\text{ is }\texttt{x} \\ \texttt{countXs(str.substring(1))} & \text{otherwise} \end{cases}$$

According to this specification, here is the implementation of `countXs`:

```
int countXs(String str)
{
      if(str.length() == 0)
      {
            return 0;
      }
      else if(str.substring(0,1).equals("x"))
      {
            return 1 + countXs(str.substring(1));
      }
      else
      {
            return countXs(str.substring(1));
      }
}
```

Note that we use `equals` to compare strings. Don't ever use == to compare strings for equality!

Task 6: Put `countXs` into your Processing sketch.

Task 7: Test `countXs` by calling it from `setup()` and printing out the results.

Task 8: Using `countXs` as a template, write `countSpaces`. Try using `countSpaces` to count the number of words in a sentence. Does this work?

Task 9: Write a recursive function `String upperX(String str)` that converts all the `x` characters in a string to upper-case. For example `upperX("xyzzyxy")` would return `"XyzzyXy"`. This function will work by considering the first character of a string and then recurring on the remainder of the string, just like `countXs`. Test your function.

Task 10: Write a recursive function `String smoosh(String str)` that removes all spaces from a string. Test your function.

Task 11: Write a recursive function `String uniq(String str)` that removes any duplicated occurrences of a letter. For example, `uniq("xyzzyy")` yields `"xyzy"` and `uniq("Hello")` yields `"Helo"`.

Task 12: Write a recursive function `float product(float[] nums, int n)` that returns the product of all numbers in the `nums` array at index `n` or greater.

Want more problems? Check out http://codingbat.com/java/Recursion-1.