Recursion in Java – Implementing Recursive Algorithms

Recursive Calculations

Suppose we are given the following recursive definition of a sequence:

$$t_1 = 2$$
$$t_n = 3t_{n-1} - 1, n > 1$$

We can express this sequence using *function notation* if we write

$$t(1) = 2$$
$$t(n) = 3t(n-1) - 1, n > 1$$

To implement such a function in Java is relatively simple. The difficult part is understanding recursion, and how the method operates.

```
public static int calculateTerm (int n)
{
      if (n < 1)
            throw new RuntimeException("Invalid parameter");
      else if (n == 1)
            return 2;
      else
            return 3 * calculateTerm(n-1) - 1;
}
```

The recursive component of this method is emphasized with **bold** type, along with the *terminating condition*. This method can be better understood by tracing its execution for a simple starting value of *n*, say n = 4.

```
original call
calculateTerm(4)
      executes
      3 * calculateTerm(3) - 1
            executes
            3 * calculateTerm(2) - 1
                  executes
                  3 * calculateTerm(1) - 1
                        returns
                        2 (terminating condition met)
                  returns
                  3 * 2 - 1 = 5
            returns
            3 * 5 - 1 = 14
      returns
      3 * 14 - 1 = 41
answer
41
```

The n^{th} term of this sequence can also be determined using a looping algorithm, rather than recursion. It is often the case that recursive algorithms can be replaced through one or more looping statements (and other logic).

```
public static int calculateTerm (int n)
{
      if (n < 1)
            throw new RuntimeException("Invalid parameter");
      else
      {
            int value = 2;
            for (int i = 2; i <= n; i++)
            {
                  value = 3 * value – 1
            }

            return value;
      }
}
```

From our previous discussion of the *greatest common divisor*, we can implement the following Java code of the recursive algorithm.

```
public static int gcd (int m, int n)
{
      if (m == n)
            return m;
      else if (m > n)
            return gcd(n, m – n);
      else
            return gcd(m, n – m);
}
```

This implementation assumes only positive, integer values for *m* and *n*. If necessary, the implementation could be adapted to negative values (perhaps using the absolute value function to force all values to be positive).

Exercises

1. Trace the execution of the `gcd` method to find the greatest common divisor of each pair of values.
   (a) m = 20 and n = 28 (b) m = 129 and n = 991

2. Trace the execution of `calculateTerm` as shown below, given an initial parameter value of 3.

```
public static int calculateTerm (int n)
{
      if (n < 0)
      {
            System.out.println("Invalid argument to method:  0 returned");
            return 0;
      }
      else if (n == 0)
            return 1;
      else
            return 2 + 0.5 * calculateTerm(n-1);
}
```

3. What is wrong with these methods?

   (a)
```
public static double bad (double a, double b)
{
      a = a/2;
      b = b*2;
      return bad(a,b);
}
```

   (b)
```
public static int badToo (int n)
{
      if (n < 1)
            return 0;
      else if (n == 1)
            return 5;
      else return 2*badToo(n+1) + 3;
}
```

4. The *factorial* of a non-negative integer *n* (written as *n!*), is a function that is useful in many branches of mathematics. It can be defined recursively as follows:

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1)! & if\ n > 0 \end{cases}$$

   (a) Write a recursive method `factorial` that returns the value of n! as a `long` value. (A `long` can represent n! for values of *n* up to 20). If an invalid parameter value is given to the method, it should print an error message and return zero.

   (b) Write a non-recursive version of `factorial`.

   (c) Which version do you think is more efficient? Justify your answer.

Recursion with Strings

Recursion can be used with both numerical and non-numerical problems. To develop algorithms for strings, we must think of a non-empty string as a recursive structure. We do this by imagining a non-empty string as a single character concatenated with a shorter (or empty) string.

For example, the string “Hello” can be viewed as:

"H" concatenated with “ello”, which is,
"e" concatenated with “llo”, which is,
"l" concatenated with “lo”, which is
"l" concatenated with “o”, which is,
"o" concatenated with “”, the empty, or null, string

To reverse the characters in a string, we could use the following algorithm.

```
To reverse a string
      if the string is not empty
            reverse the rest of the string (without the first character)
            place the first character at the end of the reversed substring
```

Using this algorithm, our original string, “Hello”, would be processed in the following way.

"Hello” is not empty, so reverse “ello” and put “H” at the end
“ello” is not empty, so reverse “llo” and put “e” at the end
“llo” is not empty, so reverse “lo” and put “l” at the end
“lo” is not empty, so reverse “o” and put “l” at the end
“” is empty, so we work ourselves back up the recursion
“o” with “l” at the end is “ol”
“ol” with “l” at the end is “oll”
“oll” with “e” at the end is “olle”
“olle” with “H” at the end is “olleH”

To implement this method in Java, we could use the following.

```
public static String reverseString (String s)
{
      if (s.length() > 0)
            return reverseString(s.substring(1)) + s.charAt(0);
      else
            return "";
}
```

Another example of recursion in strings involves finding all possible arrangements of letters (called permutations) for the given characters in a string. For example, consider the string "abcd". We can find its permutations by taking

"a" followed by all permutations of "bcd"
"b" followed by all permutations of "acd"
"c" followed by all permutations of "abd"
"d" followed by all permutations of "abc"

This process can be continued recursively until we are looking for permutations of a single character (which is obviously just that character).

To **print strings with all permutations** of a string
    if we are at the end of the string
        print the current permutation
    else
        for each character in the remaining substring
            print strings with all permutations of the substring
    end if

The Java implementation might look as follows. To print all permutations of a string, we would call this method with `index` set to zero.

```
public static void permuteString (String s, int index)
{
    String nextString;
    if (index == s.length())
    {
        System.out.println(s);
    }
    else
    {
        for (int i = index; i < s.length(); i++)
        {
           nextString = s.substring(0,index)
                        + s.charAt(i)
                        + s.substring(index,i)
                        + s.substring(i+1);

           permuteString(nextString, index+1);
        }
    }
}
```

To avoid the inconvenience of having to specify a starting value for index, we could overload this method to create a *helper method*.

```
public static void permuteString (String s)
{
    permuteString(s, 0);
}
```

Exercises

1. We previously developed an implemented an algorithm for reversing a string. Another approach to this problem is to proceed as follows:

   To **reverse** a string
   if the string is not empty
   reverse the string up to, but not including, the last character
   place the last character at the front of the string

   Write a recursive method that implements this algorithm. The method should have a header identical to the one we previously developed.

2. Yet another way to create the reversal of a string is to place the reversal of the second half in front of the reversal of the first half.

   (a) State this algorithm by completing the following, using the style shown in the previous question.

   To **reverse** a string ...

   (b) Write the recursive method that implements this algorithm. The method should have a header identical to the one we previously developed.