Understanding a recursive method

Base cases and recursive cases

Consider the definition of the nonnegative powers of 2:

$2^0 = 1$	base case
$2^k = 2*2^{k-1}$ for $k > 0$	recursive case

This definition has a *base case*: a case in which the answer is given without recursion. $2^0 = 1$.

This definition has a *recursive case*: a case in which recursion is used. $2^k = 2*2^{k-1}$. Further, note that 2^k is defined in terms of 2^{k-1}; The exponent `k-1` in the recursive definition is smaller than the exponent `k` in the thing being defined, and the base case 2^0 has the smallest exponent. This is what allows us to conclude that the definition is well-defined.

In the same way, a recursive method will have base cases (in which no recursive calls are used) and recursive cases (in which recursive calls are used). In each recursive case, the arguments of the recursive call are in some sense smaller than the parameters of the recursive method; this is what allows us to conclude that the recursion terminates.

The recursive function to the right has the base case `k = 0`, the recursive case `k > 0`, and in the recursive case the argument `k-1` of the recursive call is smaller than parameter `k`. So the recursion terminates.

```
/** = 2^k.
  * Precondition k >= 0. */
public static int pow(int k) {
   if (k == 0) return 1;
   return 2 * pow(k-1);
}
```

For a more formal treatment of termination of recursion, see item 4 under "recursion" in the JavaHyperText.

Four steps in understanding a recursive method

Step 1. Study the method's precise specification. The first step is to read the spec and gain full understanding of what the method does. If there is a precondition, read it carefully.

Don't attempt to understand a method that has no specification or a poor specification. If it has no spec, how do you know what it is supposed to do? Don't try to solve a problem unless you know what the problem is.

```
/** =  sum of the digits of n.
  * Precondition: n >= 0 */
public static int sumDigs(int n) {
   if (n < 10) return n;
   // n has at least two digits
   return n%10  + sumDigs(n/10);
}
```

Function `sumDigs`, to the right above, has a simple, short, precise specification.

Step 2. Check that the method works in the base case(s). Generally, the base cases should be written first. This is the case in function `sumDigs`.

In the case of `sumDigs`, the base case is `n < 10`. Because of the precondition, that means that $0 \leq$ `n` < 10, so `n` has one digit. And the sum of that one digit is just `n`. So the base case is correct.

Step 3. Check that the method works in the recursive case(s). This is generally the hardest of the four tasks. It requires looking at recursive calls *not* in terms of how they are executed but in terms of what they do *according to the specification*. Here's how to do it.

In your mind, replace each recursive call by what it does according to the method spec and verify that the correct result is obtained.

Let's do this for method `sumDigs`. We manipulate the expression being returned, with an example of n = 6342 to the right

```
n%10  + sumDigs(n/10)                          //2 + sumDigs(634)
   <replace sumDigs(n/10) using the method spec>
n%10  +  (sum of the digits of n/10)           //2 + sum of digits of 634
   <this is indeed the sum of the digits of n>
sum of digits of n                             // 2 + 13 = 15
```

We stress here that we do *not* try to think about how the recursive call is executed! Don't do that! Instead, just believe that the recursive call will do what the specification says it will do.

Actually, this is a proof by the technique called *mathematical induction*. If you are fluent with that technique, you will have an easier time with this.

Understanding a recursive method

Step 4. Check that the recursion terminates. The base case is when `n` < 10. In the recursive `sumDigs(n/10)`, the argument `n/10` is less than parameter `n`, so with each recursive call, the parameter decreases. For example, for the initial call `sumDigs`(6342), successive recursive calls will be `sumDigs`(634), `sumDigs`(63), and `sumDigs`(6), at which the recursion terminates.

Counting occurrences in a String

We check that function `ct`, to the right, implements its specification.

```
/** =  number of times c occurs in s */
public static int ct(char c, String s) {
   if (s.length() == 0) return 0;
   // { s has at least 1 char }
   if (s.charAt(0) != c)
       return ct(c, s.substring(1));
   // { first char of s is c}
   return 1 + ct(c, s.substring(1));
}
```

Step 1. Study `ct`'s precise specification. The spec is good. We assume that parameter `s` is not null; this is usually assumed in methods that have String parameters.

Step 2. Check that `ct` works in the base case. The base case is `s.length`() = 0, i.e. `s` is the empty String. This is the shortest possible String. In this case, character `c` occurs 0 times in `s`, and the correct value is returned.

Step 3. Check that `ct` works in the recursive cases. There are two recursive cases: `c` ≠ `s`[0] and `c` = `s`[0].

In the case that `c` ≠ `s`[0], the result should be the number of `c`'s in `s`[1..], that is, in `s.substring`(1). We check the recursive call:

`ct`(`c`, `s.substring`(1))
    <replace the call by its specification>
number of times `c` occurs in `s.substring`(1)

That is the value we expected to be returned, so this recursive call is correct.

If you can see that the returned value is correct without writing down the explicit replacement, good! Many recursive calls are simple enough for us to reason about them in our mind, without writing anything down. But if you are not sure whether the recursive call is correct, write down what the call means in terms of its specification, as we have done above.

The second recursive case, the first character is c, is handled similarly. The answer is 1 + (the number of times `c` appears in `s`[1..]). We leave the verification of correctness to you.

Step 4. Check for termination of recursion. The base case is the shortest string, with length 0. Each of the recursive calls has a smaller argument than parameter `s`, so the recursion terminates.