Another attempt at understanding recursion

We have said that the way to understand a recursive function or prove it correct is to replace each recursive call in its body by the specification of the function, with the arguments replacing parameters. We give a simple example. The math function *n factorial*, or n! is defined by

(1) 0! = 1
n! = n * (n-1) !

Here is a Java function for it.

```
/** Return n!. Precondition: 0 <= n */
public static void f(int n) {
    if (n == 0) return 1;
    return n * f(n-1);
}
```

Is this correct? We rewrite the function body, with the recursive call replaced by its spec, with the argument replacing the parameter. The replacement is shown in a different color.

```
if (n == 0) return 1;
return n * (n-1)!;
```

Looking at definition (1) above, we see that the value returned is indeed n!, so the return statement does return the correct value.

People are OK with this simple example, because the definition of n! and the method body are so similar. They have difficulty applying this method of understanding recursion to more complex recursive methods. We now attempt to explain in a different way why this works, and on the next page we'll examine another recursive method in the same way.

```
/** Return n!  Precond.: 0 ≤ n ≤ 0 */
public static f0(int n) {
    return 1;
}

/** Return n! Precond.: 0 ≤ n ≤ 1 */
public static f1(int n) {
    if (n == 0) return 1;
    return n * f0(n-1);
}

/** Return n! Precond.: 0 ≤ n ≤ 2 */
public static f2(int n) {
    if (n == 0) return 1;
    return n * f1(n-1);
}

...

/** Return n! Precond.: 0 ≤ n ≤ 99 */
public static f99(int n) {
    if (n == 0) return 1;
    return n * f98(n-1);
}

...
```

Having many functions

To the right above are a series of functions, each with a Precondition that restricts its parameter. Function f0 compute only 0! We also see that:

f1(n) computes n! for n ≤ 1 and calls f0.
f2(n) computes n! for n ≤ 2 and calls f1.
f3(n) computes n! for n ≤ 3 and calls f2.
...
f99(n) computes n! for n ≤ 99 and calls f98.

There is no recursion. Each function (except f0) calls a previous function in the list. To see that function f99 is correct, use the spec of the method it calls, f98. You verify the correctness of any of the functions in the same way.

Note also that a call like f3(3) will result in 4 frames for calls being placed on the call stack, as shown to the right.

```
frame for f0(0)
frame for f1(1)
frame for f2(2)
frame for f3(3)
```

You can think of recursive function f below as simply an abbreviation of that long list of functions. And you can verify its correctness just as we did when dealing with the 100 functions f(1), f(2), ... f(99): Replace the recursive call f(n-1) by the spec of f, with the parameter replaced by the argument. That is replace f(n-1) by (n-1)!. You also know that for a call f(3), at one point the call stack will be as shown to the right.

```
frame for f(0)
frame for f(1)
frame for f(2)
frame for f(3)
```

If it helps, think of the recursive call f(n-1) as a call on *another* function with the same kind of body, and understand the recursive call in terms of its specification.

```
/** Return n! Precond.: 0 ≤ n */
public static f(int n) {
    if (n == 0) return 1;
    return n * f(n-1);
}
```

Another attempt at understanding recursion

What about merge sort?

We now treat recursive procedure mergesort the same way. In order to keep the explanation simple, we work only with array segments b[h..k] whose length is a power of 2.

To the right, we have procedure ms1, whose precondition requires that b[h..k] have size 0 or 1. It simply returns.

Procedure ms2 sorts an array segment of size 0, 1, or 2. If its size is 2, it calls ms1 twice, each time with an array segment of size 1. You know that ms1 is correct, and you rely on its specification to verify that ms2 is correct.

Procedure ms4 sorts an array segment of size 0, 1, 2, or 4. If its size is 2 or 4, it calls ms2 twice, each time with an array segment of half the size. You know that ms2 is correct, and you rely on its specification to verify that ms2 is correct.

And so on. We show method ms64. It sorts an array segment of size 0, 1, 2, 4, 8, 16, 32, or 64. If its size is 2, 4, 8, 16, 32, or 64, it calls ms32 twice, each time with an array segment of half the size. You know that ms32 is correct, and you rely on its specification to verify that ms64 is correct.

Look how similar the bodies of ms2, ms4, ms8, ..., ms64 are. The only difference is the two calls on a previous procedure, for e.g. ms64 calls ms32 twice. Thus, instead, we write one recursive procedure, eliminating the digits in the name of the method:

```
/** Sort b[h..k].
  Precond.: 0 <= k+1-h and is a power of 2 */
public static ms(int[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    ms(b, h, e);
    ms(b, e+1, k);
    merge(b, h, e, k); }
```

If it helps you, think of a recursive call like ms(b, h, e) as a call on *another* procedure that looks exactly the same, and rely on its specification —Sort b[h..k]— in understanding what the call does.

Finally, to the right, we show on the left the call stack at one point in executing ms64(b, 0, 7) and to its right the call stack in calling the method shown above with ms(b, 0, 7).

```
/** Sort b[h..k].
  Precond.: 0 <= k+1-h <= 1 */
public static ms1(int[] b, int h, int k) {
    return;
}

/** Sort b[h..k].
  Precond.: k+1-h <= 2 and is a power of 2 */
public static ms2(int[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    ms1(b, h, e);
    ms1(b, e+1, k);
    merge(b, h, e, k); }


/** Sort b[h..k].
  Precond.: k+1-h <= 4 and is a power of 2 */
public static ms4(int[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    ms2(b, h, e);
    ms2(b, e+1, k);
    merge(b, h, e, k); }

...

/** Sort b[h..k].
  Precond.: k+1-h <= 64 and is a power of 2 */
public static ms64(int[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    ms32(b, h, e);
    ms32(b, e+1, k);
    merge(b, h, e, k); }

...
```

Using many methods	**Using recursion**
frame for ms8(b, 0, 0)	frame for ms(b, 0, 0)
frame for ms16(b, 0, 1)	frame for ms(b, 0, 1)
frame for ms32(b, 0, 3)	frame for ms(b, 0, 3)
frame for ms64(b, 0, 7)	frame for ms(b, 0, 7)

Another attempt at understanding recursion