# Recursion may require extra parameters

Consider writing a recursive procedure to sort an array `b` —elsewhere in this JavaHyperText we show two such recursive procedures, quicksort and mergesort. We can't just use

```
/** Sort b. */
public static void sort(int[] b)
```

because there is no way to write a recursive call with an argument that is "smaller" than `b`. Instead, we generally add two extra parameters and write

```
/** Sort b[h..k] */
public static void sort(int[] b, int h, int k)
```

We could, if we wanted, have both procedures, as shown to the right, with the one-parameter `sort` calling the recursive `sort`. This is standard practice.

```
/** sort b */
public static void sort(int[] b) {
   sort(b, 0, b.length-1);
}

/** sort b[h..k] */
public static void sort(int[] b, int h, int k) {
   …
}
```

### Making recursive methods on strings more efficient

In introducing recursion, we wrote function `ct`, on the right, to calculate the number of times a character `c` appears in a string `s`. This method is far too inefficient, taking time proportional to the square of the length of `s`. This is because the substring operation takes time proportional to the length of the substring —a new String object has to be created and the substring copied into it.

```
/** =  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));
}
```

To get a more efficient recursive function, write a function `ct` with three parameters, as shown below, and change the two-parameter function `ct` to call the three-parameter one. Recursive function `ct (c, s, h)` does not use the expensive substring operation, and the time it takes is proportional to the length of `s`.

```
/** =  number of times c occurs in s */
public static int ct(char c, String s)
   { return ct(c, s, 0); }

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

The three-parameter function `ct` is harder to read and understand than the original one, but the increased efficiency is worth it.

Since we are talking about efficiency, we also add that an iterative implementation of `ct` would be even better because there would be no recursive calls at all. We wrote `ct` recursively only to help introduce recursion, and the iterative version is preferred.

### An exercise for you

Function `isPal`, to the right below, determines whether its parameter `s` is a palindrome —whether it reads the same backward and forward. Examples of palindromes are "" (the empty string), "n", and "noon".

Function `isPal` is inefficient because it uses the substring operation. Rewrite it using extra parameters, as we did above.

```
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1) return true;

    // { s has at least 2 chars }
    int n= s.length()-1;
    return s.charAt(0) == s.charAt(n)  &&
           isPal(s.substring(1, n));
}
```