The call `power(5, 3)` would be executed as follows:

```
power(5, 3) finds that 3 is not equal to 0, so it calls
  power(5, 2), which finds that 2 is not equal to 0, so it calls
    power(5, 1), which finds that 1 is not equal to 0, so it calls
      power(5, 0), which finds that 0 is equal to 0, so it returns 1, causing
    power(5, 1) to return 5 × 1 = 5, causing
  power(5, 2) to return 5 × 5 = 25, causing
power(5, 3) to return 5 × 25 = 125.
```

Incidentally, we can condense the `power` function a bit by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

Both `fact` and `power` are careful to test a "termination condition" as soon as they're called. When `fact` is called, it immediately checks whether its parameter is less than or equal to 1. When `power` is called, it first checks whether its second parameter is equal to 0. All recursive functions need some kind of termination condition in order to prevent infinite recursion.

The Quicksort Algorithm

At this point, you may wonder why we're bothering with recursion; after all, neither `fact` nor `power` really needs it. Well, you've got a point. Neither function makes much of a case for recursion, because each calls itself just once. Recursion is much more helpful for sophisticated algorithms that require a function to call itself two or more times.

In practice, recursion often arises naturally as a result of an algorithm design technique known as ***divide-and-conquer***, in which a large problem is divided into smaller pieces that are then tackled by the same algorithm. A classic example of the divide-and-conquer strategy can be found in the popular sorting algorithm known as ***Quicksort***. The Quicksort algorithm goes as follows (for simplicity, we'll assume that the array being sorted is indexed from 1 to n):

1. Choose an array element e (the "partitioning element"), then rearrange the array so that elements $1, \ldots, i-1$ are less than or equal to e, element i contains e, and elements $i+1, \ldots, n$ are greater than or equal to e.
2. Sort elements $1, \ldots, i-1$ by using Quicksort recursively.
3. Sort elements $i+1, \ldots, n$ by using Quicksort recursively.

After step 1, the element e is in its proper location. Since the elements to the left of e are all less than or equal to it, they'll be in their proper places once they've been sorted in step 2; similar reasoning applies to the elements to the right of e.

Step 1 of the Quicksort algorithm is obviously critical. There are various methods to partition an array, some much better than others. We'll use a technique