Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-three)

3.4: Recursion

Certain types of problems, especially those having to do with sorting and searching, can often conveniently be formulated in terms of simple functions that, to accomplish their goal, call themselves. These so-called *recursive* functions accomplish a kind of divide-and-conquer strategy: each time the function is called, it simplifies the problem into subproblems, then calls itself to solve the subproblems. Eventually the subproblems become solvable by inspection; the recursion then ends. Every recursive function must provide some kind of "escape clause" so that the recursion does not continue forever.

Sorting an array of numbers is an operation for which you can usefully implement recursion. One common sorting algorithm, known as "quicksort," is easy to define recursively. It can be described by the following procedure:

1. Randomly select one element of the array, the "pivot."
2. Divide the rest of the array into two subarrays: those elements that are greater than the pivot, and those that are less than or equal to it.
3. Quicksort each subarray that contains more than one element.
4. Return an array consisting of the elements from the less-than subarray, the pivot, and the elements from the greater-than subarray.

As you can see, the third step is a recursive call of the whole quicksort procedure. The recursion will stop when a subarray has fewer than two members.

For the recursive implementation of quicksort to work, the variables that hold and track the subarrays must be local to each invocation of the function. Each invocation will need its own copies, which it can change without affecting any of the other copies.

The main reason I mention recursion is not particularly the prevalence of *directly* recursive functions (such as our quicksort implementation) but rather the surprising prevalence of *indirect* recursion. In complex software systems, many functions that never intended to call themselves directly can end up doing so indirectly; functions called by a function can call other functions which eventually call the original function! For this reason, it is important to minimize the reliance on global data of all functions.