KIT101 Notes — 17 — Recursion

James Montgomery

Revised: 2018-02-16

References: L&L 12.1–12.4

Introducing Recursion

Consider a матрёшка (matryoshka) doll

A matryoshka doll, often called a “Russian doll”, is a children’s toy made up of a series of nested hollow dolls, except for the innermost doll, which is solid. Although there are typically at most eight levels of doll, one could imagine it going on further, with progressively larger and larger dolls around the outside. We can define the doll like this:

A **matryoshka doll** (“Russian doll”) is
- a small solid wooden doll
- or a hollow wooden doll containing a **matryoska doll**

A list is...

We can do the same thing for non-tangible things like lists. Consider this list of numbers: 24, 88, 40, 37

We can define the way this list is written as:
A **list** is:
- a number (for example, 37)
- or a number comma **list** (for example, 40, 37; or 88, 40, 37; or 24, 88, 40, 37)

In both cases the definition contains the thing that is being defined (but also in both cases we can see that the nested item (whether it's a doll or a list) is getting smaller each time, until it reaches a base case that is trivially defined.

Recursion as a problem solving technique

The approach when using recursion as a problem solving technique is to divide a problem into

- one "step" that makes the problem smaller (but of the same type)
- a base case (where the solution is trivial)

The way this is *implemented* in a program is by having a (recursive) method call itself with different arguments (that describe a "smaller" problem). When the arguments define the base case the method no longer calls itself.

Recursive methods and the stack

A method can call itself because each method is really a set of *reusable* instructions. Each invocation (also called an 'activation') of a method is allocated some space on the call stack to hold the values of its parameters and local variables. If a method calls itself then its current execution point is saved and a new 'stack frame' is placed on the stack for the new call. When that call ends execution resumes in the first 'activation' of the method where it left off.

The general shape of a recursive solution

The general pattern for a recursive solution is

1. test for stopping condition
2. If not at the stopping condition, either
    - do one step towards solution
    - and call the method again to solve the rest
      OR
    - call the method again to solve most of the problem
    - and do the final step

The first alternative occurs when we want to *do* something at each step, such as modifying each element in an array. The second alternative occurs frequently when we want to calculate some result and each step depends on us determining an answer to the rest of the problem first.

Note that recursion must stop in order to be useful, so the *base case* must be carefully defined so that the method will eventually stop calling itself (otherwise you'll get a `StackOverflowError`).

Devising recursive solutions

1. Find a case where the solution is trivial (the stopping condition)
2. Divide the problem up:
    - One step & a smaller problem of exactly the same type (closer to the trivial solution)

3. Believe that the solution will work (must imagine that the recursive call has done all the necessary work for the *present* call to use its result)
4. Code and test the solution

Examples

Recursion is easy to define, but takes practice to implement. The following examples illustrate how some problems can be broken down into recursive solutions.

Binary search is conceptually recursive

Although we have previously implemented binary search iteratively (indeed, *anything* that can be implemented using a loop can be done using recursion, and the same in reverse), it is conceptually recursive. Each sub-problem is defined by the region of the array in which to search (starting with the whole array).

- Binary search has two base cases:
    1. Have no more elements to search: not found
    2. Middle element is the target: found
- Recursive case: binary search the likely half of the array

For Your Interest: An implementation of recursive binary search

Below is the iterative implementation of binary search from Week 10, next to an equivalent recursive implementation. This also demonstrates a common pattern when implementing recursive solutions in object-oriented languages: a public method that is equivalent to the original, iterative version, that calls the (private) recursive implementation (which has more parameters and so would be less convenient to call directly). Note how some of the local variables from the iterative version become parameters in the recursive version.

Iterative version	Recursive version

`public static int binarySearch(int[] a, int t) {`
`  boolean found = false;`
`  int low = 0;`
`  int high = a.length - 1;`
`  int middle;`
`  int index = -1;`
 	
`  while (low <= high && !found) {`
`  middle = (low + high)/2;`
`  if (a[middle] ==`
	`/** The public interface to recursive binary search. */`
	`public static int recursiveBinarySearch(int[] a, int t) {`
	`  return recursiveBinarySearch(a, t, 0, a.length - 1);`
	`}`
	
	`/** The recursive implementation. */`
	`private static int recursiveBinarySearch(int[] a, int t, int low, int high) {`
	`  int pos; //result of *this* call to binarySearch`
	`  int middle;`

```
t) {
                found =
true;
                index =
middle;
            } else {
                if (t <
a[middle]) {
                    high =
middle - 1;
                } else {
                    low =
middle + 1;
                }
            }
        }
        return index;
}
```

```
    if (low > high) { //base case 1: not
found
            pos = -1;
        } else {
            middle = (low + high) / 2;
            if (a[middle] == x) { //base case
2: found
                pos = mid;
            } else if (a[middle] < x) {
//recursive case 1: search top half
                pos = binarySearch(a, x,
middle+1, high);
            } else { //recursive case 2:
search bottom half
                pos = binarySearch(a, x, low,
middle-1);
            }
        }
        return pos;
}
```

Sum of the first *n* integers

Ignoring the fact that there is an equation to solve this problem, calculating the sum of the first *n* positive integers can be done recursively.

The base case is the smallest possible value of *n*, which is 1: sum(1) = 1.

The recursive case is to add the *current* value of *n* to the result of sum(*n* – 1). The code for this is quite compact:

```
public int sum(int n) {
    int total;
    if (n == 1) { //base case
        total = 1;
    } else { //recursive case
        total = n + sum(n – 1);
    }
    return total;
}
```

Factorial of *n*

The factorial (see https://en.wikipedia.org/wiki/Factorial) of a non-negative integer *n* is the product of all positive integers up to *n*. It is normally denoted *n*!. It has various applications including determining the number of ways a given set of items may be ordered (arranged). Most definitions of factorial define 0! = 1, which gives us the following recursive definition:

- 0! = 1

- $n! = n \times (n - 1)!$

We can implement this in a method as follows:

```
public int fact(int n) {
    if (n == 0) { //base case
        return 1;
    } else { //recursive case
        return n * fact(n - 1);
    }
}
```

Advantages and disadvantages of recursion

- Advantages
  - Some problems have complicated iterative solutions, but conceptually simple recursive ones
  - Good for dealing with dynamic data structures (where the size is determined at run time)
- Disadvantages
  - Extra method calls use memory space & other resources
  - Thinking up recursive solution is hard at first
  - Might not *look* like a recursive solution will work
