Recursion in Java

Recursion Defined

Recursion is a technique for defining data structures or algorithms *in terms of themselves*. A **recursive algorithm** is a form of decomposition where rather than choosing an arbitrary subtask of the problem to do, choose a simpler problem that has *the same form* as the original (self-similarity). A recursive definition has two parts:

– the **base case** - a stopping condition
– the **recursive step** - an expression of the computation or definition in terms of itself

There may be one or more of each of these.

Example of a Recursive Definition

- There are many recursive definitions in mathematics. Consider the factorial function:

  n! = n * (n-1) * (n -2) * … * 2 * 1

- The same function can be defined recursively by giving a base case and a recursive step:

  0! = 1 (by definition)
  n! = n * (n - 1)! (the recursive step)

Recursive Functions

A **recursive function** is a function which calls itself somewhere in the function body. Recursive functions are supported in most modern programming.

A language that supports recursion usually requires a system stack for tracking function call and return.

In a recursive function, execution must “drive” computation to a base case so the recursion will stop. The recursive step is intended to ensure that the computation eventually terminates.

The Factorial Function in Java

The recursive definition for factorial can be written in a very straightforward manner. Take some time to convince yourself that this method works:

```
// precondition: n >= 0
public static int fact(int n)
{
   Assert.pre(n >= 0);
   if (n < 2)      // base case
     return 1;
   else             // recursive step (call in bold)
     return (n * fact(n - 1));
}
```

Tracing a Recursive Function

The behavior of fact when n = 5:

```
fact(5) -> 5 * fact(4)
fact(4) -> 4 * fact(3)
fact(3) -> 3 * fact(2)
fact(2) -> 2 * fact(1)
fact(1) -> 1
```

We can view this as a process of **driving the computation to the base case**; next, we *unwind.*

Tracing a Recursive Function

Eventually, the call **fact(1)** returns 1 to the function which called it, so that it can complete its calculation and return its result to the function which called *it*, and so on. This is called *back-substitution*, or *unwinding* the recursion.

```
fact(1) -> 1
fact(2) -> 2 * 1 = 2
fact(3) -> 3 * 2 = 6
fact(4) -> 4 * 6 = 24
fact(5) -> 5 * 24 = 120 <<< the final answer
```

Recursion Under the Hood

The execution environment sets up a *call stack* (or *system stack)* to store activation records for keeping track of function call and return. Therefore, each recursive call is represented by an activation record on the stack. The activation record at the top of the stack is active, and all other calls on the stack are said to be *suspended*. Each recursive-step call waits for the results of the next call so it can finish its own computation.

In geeneral, recursion works by taking advantage of the system stack to keep track of the *partial results* computed by the successive recursive calls; these partial results are then back-substituted into the preceding calls through the normal operation of return values.

Another Mathematical Example

An important function in probability is the **binomial coefficient**, or **choose**, function, written **C(n,k)**, and defined as:

$$C(n,k) = n! / [k!(n - k)!]$$

This is hard to compute because n! gets too large to represent as an integer even for small values of n. Another version is recursive (note: two base cases):

$C(n, 0) = 1$, for $n > 0$
$C(n, n) = 1$, for $n > 0$
$C(n, k) = C(n-1, k) + C(n-1, k-1)$, for $n > k >= 0$

The choose function in Java

Here is a straightforward translation of the recursive definition into Java:

```
// pre: n > 0 && n > k >= 0
public static int choose(int n, int k)
{
   if (k == 0)        // first base case
      return 1;
   else if (n == k)   // second base case
      return 1;
   else               // recursive step
      return choose(n-1, k) + choose(n-1, k-1);
}
```

Recursion Trees

A trace is one method of analyzing what a recursive function is doing. Another is to draw a *recursion tree*. In this method, we show each invocation of the function as a *tree node* and draw lines between each function invocation and the recursive calls it makes.

The recursion tree can be annotated to show arguments to each function and the values they compute and return.   The tree for `fact` gives us no new information, but drawing such a tree for the `choose` function is quite useful.

Recursion tree for choose()

We derive the *recursion tree* for `choose(4, 2);`

Another example

```
// pre: data[] is sorted
// post: returns index if target in data, -1 otherwise
public static int binarySearch(int[] data, int first,
                                   int last, int target)
{
  if (first > last) // base case #1
      return -1;

  int mid = (first+last)/2;

  if (data[mid] == target)  // base case #2
    return mid;
  else if (data[mid] > target) // rec step #1
    return binarySearch(data, first, mid-1, target);
  else
   return binarySearch(data, mid+1, last, target);
}
```

Types of Recursion

Each of these simple examples illustrates a different kind of recursion:

- The factorial and binarySearch methods are examples of *linear recursion*, in which only one recursive call is made per recursive step.
- The choose method is an example of *tree recursion*, in which two (or more, in general) recursive calls are made in at least one recursive step.

The recursion tree for choose exposes an inefficiency: some recursive calls do redundant work (though this is not always true for tree-recursive algorithms).

Recursion vs. Iteration

For these simple examples, it is easy to come up with an iterative version of the same algorithm that will run faster. In fact recursion and iteration are related; languages without iteration simulate it using recursion, and vice versa.

However, more interesting examples, such as the sorting routines we discuss later, do *not* have an obvious iterative solution. When a solution is discovered, it is often much longer than the recursive version.

The same is true for algorithms based on recursively-defined data structures, such as the Binary Tree ADT.

Efficiency of Recursive Algorithms

The *efficiency* of a recursive algorithm is not obvious and depends on the type and number of recursive calls performed. Methods used to determine efficiency include:

- estimating number of operations done by counting the number of recursive calls from the trace or recursion tree and multiplying by the number of operations per call.
- mathematically by using *recurrence relations* to model the performance of the function for any given input (usually studied in a discrete mathematics class).
- Some functions require even more sophisticated mathematical tools to analyze.

Space Complexity

Space complexity is a characterization of how much memory an algorithm requires as a function of input size. We can generally give an upper bound on the amount of memory any particular function will use.

Many algorithms use a constant amount of space; for example, most sorting algorithms manipulate the values of an array in place, so the total amount of space used does not change *during execution*.

Recursive methods save their state on the *system stack*, which is a bounded resource, so we **must** consider space complexity for these types of algorithms.

Examples of Space Complexity

- The fact method as written makes **n** total calls to perform fact(n), therefore, at most **n** activation records will be pushed on the stack.
- The binarySearch method makes a number of calls equal to or less than the log (base 2) of the size of the input array, so no more than $\log_2$`(data.length)` activation records will be pushed. In this case we may be able to do even better.
- To figure space complexity of the choose method, notice from the recursion tree that any path is made up of at most **n** calls (including the first). This means that at most **n** activation records will be on the stack at any given time during execution**.**

Space Complexity: Conclusions

The space used by the recursive functions fact and choose (in terms of activation records on the stack) is proportional to the argument **n**, so we say that these functions have *linear space complexity*.

The space used by the recursive function binarySearch is proportional to the logarithm of the array size, so it has *logarithmic space complexity* expressed in terms of the number of array elements.

Tail Recursion

Tail recursion is defined as a recursive function that returns immediately after its recursive step - it does no computation with a result from a previous call.

In fact and choose, results of one recursive call are returned to the calling function and used in a calculation; such functions are **not** tail-recursive.

By contrast, binarySearch returns the value produced by the base case call, so there are no partial results to compute, and all back-substitutions are unnecessary. This is a **tail-recursive function**.

Tail Call Optimization

The programmer or the compiler may make use of the tail-recursion property. If a function is tail recursive, it can be easily turned into an iterative function by the coder. However, some compilers automatically recognize this case and optimize the code for us.

This amounts to reducing the space required to a **constant amount** by reusing the original call's activation record. The compiler can do this because a tail recursive function call saves no partial results in the record.

Summary

- Many mathematical definitions and data structures are naturally **recursive**, as are many patterns observed in nature.
- Recursion can be expressed using **recursive methods** which have at least one *base case* along with at least one *recursive call*.
- Simple recursive methods may be slower and use more memory than equivalent iterative versions. However, we will see examples for which the recursive version is shorter, faster, and easier to understand.