CSE 1030

Yves Lespérance

Lecture Notes

Week 8 — Recursion

Recommended Readings:
Van Breugel & Roumani Ch. 7 and Savitch Ch. 11 and Sec. 12.2

Recursive Methods

Consider the factorial function in maths:

$0! = 1$
$1! = 1$
$2! = 2 \times 1 = 2$
$3! = 3 \times 2 \times 1 = 6$
$4! = 4 \times 3 \times 2 \times 1 = 24$
$\ldots$
$n! = n \times (n-1) \times \ldots \times 1 \qquad (*)$

A simple Java method that uses a loop to compute the function is:

```
public static int factorial(int n)
{  int prod = 1;
   for(int i = 1; i <= n; i++)
      prod = prod * i;
   return (prod);
}
```

In mathematics, it is common to give a *recursive* definition to such functions:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

Notice that the function is mentioned on the right hand side of the definition! Yet it is not circular; we can use it to find the value of the function for any argument, e.g.:

$4! = 4 \times 3!$
$\quad 3! = 3 \times 2!$
$\qquad 2! = 2 \times 1!$
$\qquad\quad 1! = 1 \times 0!$
$\qquad\qquad 0! = 1$
$\qquad\quad 1! = 1 \times 1 = 1$
$\qquad 2! = 2 \times 1 = 2$
$\quad 3! = 3 \times 2 = 6$
$4! = 4 \times 6 = 24$

In contrast to (*), this definition precisely specifies $n!$ for arbitrarily large n's. Recursive definitions are also called inductive definitions.

This kind of approach is often used in computer algorithms and programs. We can write a Java method that computes $n!$ by recursion:

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

Here is a *call trace* of `factorial(4)`:

```
factorial(4) calls factorial(3)
  factorial(3) calls factorial(2)
    factorial(2) calls factorial(1)
      factorial(1) calls factorial(0)
        factorial(0) returns 1
      factorial(1) returns 1 * factorial(0) i.e. 1
    factorial(2) returns 2 * factorial(1) i.e. 2
  factorial(3) returns 3 * factorial(2) i.e. 6
factorial(4) returns 4 * factorial(3) i.e. 24
```

This is implemented using an execution stack which keeps track of what methods are called, what the values of their parameters/local variables are, and where the methods will resume.

With recursive methods, the execution stack will contain several entries for the same method, e.g. the recursive calls to `factorial`.

When recursion is not allowed (as in Fortran), there can never be more than one call of a method that is active. So storage for the local variables of methods can be allocated at compile time (statically). But when recursion is allowed as in all modern languages , a stack must be used to accomodate an arbitrary number of calls.

Recursion: Definitions

A procedure/method is *recursive* iff it calls itself from within its own body either directly or indirectly (e.g. method `m1` calls method `m2` which calls `m1`).

A recursive solution to a problem involves two components:

1. a direct solution to some simple instances of the problem; these are called *base cases*;

2. a solution to the general case of the problem that involves solving a *simpler* instance(s) of the problem and performing some operations on the result; this is called the *recursive case*.

The measure of problem size you are using is what you are doing *recursion over*; e.g. for factorial, it is n; first thing you need for designing a recursive solution.

Proof of Correctness by Induction: E.g. factorial

Show base case is correct: `factorial(0)` returns 1 which is 0!; correct.

For recursive case, assume the recursive calls, if they terminate, are correct, and prove that the recursive case is correct given this assumption.

So assume `factorial(n-1)` (if it terminates) correctly returns $(n-1)!$.

Then `factorial(n)` returns `n * factorial(n-1)`.

Since `factorial(n-1)` is correct by our assumption, `n * factorial(n-1)` = $n * (n-1)!$, which is $n!$.

So the recursive case is correct and thus the method is correct.

Proof of Termination: E.g. factorial

Define the size of each invocation of the method. This must be a natural number.

Then show that each recursive invocation has a smaller size than the original invocation.

If this is the case the method must terminate.

For factorial:

Let $size(\texttt{factorial(n)}) = \texttt{n}$.

`factorial(n)` only makes the recursive call `factorial(n-1)`.

$size(\texttt{factorial(n-1)}) = (\texttt{n} - 1) < size(\texttt{factorial(n)}) = \texttt{n}$.

So the method terminates.

A Combined Proof by Induction of Correctness and Termination: E.g. factorial

We prove correctness and termination of the factorial method by induction on the value of the argument `n`.

1) Base case $\texttt{n} = 0$: then `factorial(0)` terminates and returns $1 = 0!$, which is correct.

2) For the recursive case: Assume that `factorial(n)` is correct and terminates for all $\texttt{n} \leq k$. This is the *induction hypothesis*.

We must prove that that the method is correct for $\texttt{n} = k + 1$.

Then `factorial(n)` returns $(k + 1) * \texttt{factorial(}k\texttt{)}$.

Since $\texttt{factorial(}k\texttt{)}$ is correct by our assumption, $(k+1) * \texttt{factorial(}k\texttt{)}$ $= (k + 1) * k! = (k + 1)!$.

So for all natural numbers n $\texttt{factorial(}n\texttt{)}$ is correct and terminates.

E.g. Computing Terms of Fibonacci Sequence Recursively

The Fibonacci sequence can given a recursive definition as follows:

$$fibo(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fibo(n-1) + fibo(n-2) & \text{otherwise} \end{cases}$$

This can be translated directly into a recursive Java method:

```
public static int fibo(int n)
{  if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fibo(n - 1) + fibo(n - 2);
}
```

Call tree for `fibo(5)`:

[Draw on board]

Call trace for `fibo(5)`:

```
fibo(5) calls fibo(4)
  fibo(4) calls fibo(3)
    fibo(3) calls fibo(2)
      fibo(2) calls fibo(1)
        fibo(1) returns 1
      fibo(2) calls fibo(0)
        fibo(0) returns 0
      fibo(2) returns fibo(1) + fibo(0) i.e. 1
    fibo(3) calls fibo(1)
      fibo(1) returns 1
    fibo(3) returns fibo(2) + fibo(1) i.e. 2
  fibo(4) calls fibo(2)
    fibo(2) calls fibo(1)
      fibo(1) returns 1
    fibo(2) calls fibo(0)
      fibo(0) returns 0
    fibo(2) returns fibo(1) + fibo(0) i.e. 1
  fibo(4) returns fibo(3) + fibo(2) i.e. 3
```

```
fibo(5) calls fibo(3)
  fibo(3) calls fibo(2)
    fibo(2) calls fibo(1)
      fibo(1) returns 1
    fibo(2) calls fibo(0)
      fibo(0) returns 0
    fibo(2) returns fibo(1) + fibo(0) i.e. 1
  fibo(3) calls fibo(1)
    fibo(1) returns 1
  fibo(3) returns fibo(2) + fibo(1) i.e. 2
fibo(5) returns fibo(4) + fibo(3) i.e. 5
```

Problem: Write a recursive version of a method `isPalindrome(String word)` that returns `true` iff `word` is a palindrome, i.e. reads the same backwards and forwards; e.g. `noon` and `dad` are palindromes.

Recursive Methods for List Processing

E.g. a method `contains(T element, List<T> list)` that searches `list` to see if `element` appears in it.

Note that version below is more efficient than Van Breugel and Roumani's because doesn't make copy of rest of list.

```
public static <T> boolean contains(T element, List<T> list)
{
   boolean contains;
   if (list.size() == 0)
   {
      contains = false;
   }
   else
   {
      if (element.equals(list.get(0)))
      {
         contains = true;
      }
      else
      {
         List<T> rest = list.subList(1, list.size());
         contains = MyClass.contains(element, rest);
      }
   }
   return contains;
}
```

Proof of Correctness by Induction:

Base case `list.size()` = 0: Correct, since `element` cannot be contained in an empty list.

For recursive case `list.size()` ≥ 1:

Induction hypothesis: Assume the recursive call, if it terminates, is correct.

`list` has at least one element. If `element` is equal to the first element of `list` we return true. Correct.

If `element` is not equal to the first element of `list`, it is contained in `list` iff it is contained in the rest of the list.

In this case, the method return the result of a recursive call on the rest of the list. By the induction hypotesis, this recursive call is correct (if it terminates).

So if it terminates, the method returns the correct answer in the recursive case.

Proof of Termination

Let *size*(`contains(element, list)`) = `list.size()`.

This is a natural number.

Show size of recursive invocation is smaller.

`list.subList(1, list.size()).size()` < `list.size()`

So *size*(`contains(element, rest)`) < *size*(`contains(element, list)`).

So the method terminates.

See other list processing examples in Van Breugel and Roumani.

Pitfalls of Recursion

One pitfall of recursion is *infinite regress*, i.e. a chain of recursive calls that never stops. E.g. if you forget the base case "`n == 0`" in `factorial`. Make sure you have enough base cases.

Anything that can be done using a loop can be done by recursion.

However, recursion can be less efficient than an iterative implementation. This can happen because there is recalculation of intermediate results as in `fibo`. Or there can be extra memory use because recursive calls must be stored on the execution stack, e.g. `factorial`.

Advantages of Recursion

But some algorithms can be coded much more simply using recursion and there is no loss of efficiency. E.g., traversing a tree and printing the labels on the nodes. Another e.g.: Mergesort algorithm to sort an array/collection.

Also some types of recursion do not require the use of additional stack memory and good compilers can take advantage of this. These techniques are studied in "functional programming" .