CSE 1030

Yves Lespérance

Lecture Notes

Week 10 — Recursion

Recommended Readings:
Van Breugel & Roumani Ch. 8 and Savitch Ch. 11 and Sec. 12.2

Recursive Methods

Consider the factorial function in maths:

$$
\begin{aligned}
&0! = 1\\
&1! = 1\\
&2! = 2 \times 1 = 2\\
&3! = 3 \times 2 \times 1 = 6\\
&4! = 4 \times 3 \times 2 \times 1 = 24\\
&\ldots\\
&n! = n \times (n-1) \times \ldots \times 1 \qquad (*)
\end{aligned}
$$

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.:

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

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 `0` which is `0!`; correct.

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

So assume `factorial(n-1)` correctly returns $(\mathtt{n}-1)!$.

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

Since `factorial(n-1)` is correct by our assumption, `n * factorial(n-1)` = $\mathtt{n} * (\mathtt{n}-1)!$, 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. Must be a natural number.

Let $size($`factorial(n)`$) =$ `n`.

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

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

$size($`factorial(n-1)`$) = ($`n` $- 1) < size($`factorial(n)`$) =$ `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 `n` $= 0$: then `factorial(0)` returns $0 = 0!$; correct.

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

We must prove that that the method is correct for `n` $= k + 1$.

Then `factorial(n)` returns $(k + 1)*$`factorial(`k`)`.

Since `factorial(`k`)` is correct by our assumption, $(k+1)*$`factorial(`k`)` $= (k + 1) * k! = (k + 1)!$.

So for all natural numbers n `factorial(`n`)` 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
```


Sorting

Sorting an array or list is a very common operation. The array/list is sorted if all its elements appear in the right order.

What “right order” means is application dependent. E.g. sort array of Student objects so that they appear in increasing order of student number. Could also sort by names using lexicographic order. Another e.g.: could sort marks in a class in decreasing order.

It is much easier to find an item if the array/list is sorted than if it is not (e.g. in a phone book).

There are many different algorithms for array sorting. Much work has been done to analyse them and determine which are the best in terms of running time.

Merge Sort — A Recursive Array Sorting Algorithm

Steps:

1. divide the array into 2 halves;
2. sort each half recursively;
3. merge the sorted halves back into a single array.

```
public static void mergeSort(int[] a, int from, int to)
{  if(from == to) return;
   int mid = (from + to) / 2;
   // sort both halves recursively and merge back
   mergeSort(a, from, mid);
   mergeSort(a, mid + 1, to);
   merge(a, from, mid, to);
}

public static void sort(int[] a)
{  mergeSort(a, 0, a.length - 1);
}
```



```
public static void merge(int[] a, int from, int mid, int to)
{  // create temporary array
   int both = to - from + 1;
   int[] tempA = new int[both];
   // merge until one array runs out
   int i1 = from;
   int i2 = mid + 1;
   int j = 0;
   while(i1 <= mid && i2 <= to)
   {  if (a[i1] < a[i2])
      {  tempA[j] = a[i1];
         i1++;
      }
      else
      {  tempA[j] = a[i2];
         i2++;
      }
      j++;
   }
```



```
   // copy rest of remaining half
   while(i1 <= mid){
      tempA[j] = a[i1];
      i1++;
      j++;
   }
   while(i2 <= to){
      tempA[j] = a[i2];
      i2++;
      j++;
   }
   // copy tempA back into a
   for (j = 0; j < both; j++)
      a[from + j] = tempA[j];
}
```


Tracing execution of Merge Sort on `a = [ 18, 33, 4, 21, 17, 35, 20 ]`

```
mergeSort(a,0, 6) calls mergeSort(a,0,3)
  mergeSort(a,0,3) calls mergeSort(a,0,1)
    mergeSort(a,0,1) calls mergeSort(a,0,0)
      mergeSort(a,0,0) returns
    mergeSort(a,0,1) calls mergeSort(a,1,1)
      mergeSort(a,1,1) returns
    mergeSort(a,0,1) calls merge(a,0,0,1)
      merge(a,0,0,1) merges [18] and [33] returns with a unchanged
    mergeSort(a,0,1) returns with a unchanged
  mergeSort(a,0,3) calls mergeSort(a,2,3)
    mergeSort(a,2,3) calls mergeSort(a,2,2)
      mergeSort(a,2,2) returns
    mergeSort(a,2,3) calls mergeSort(a,3,3)
      mergeSort(a,3,3) returns
    mergeSort(a,2,3) calls merge(a,2,2,3)
      merge(a,2,2,3) merges [4] and [21] returns with a unchanged
    mergeSort(a,2,3) returns with a unchanged
  mergeSort(a,0,3) calls merge(a,0,1,3)
    merge(a,0,1,3) merges [18, 33] and [4, 21]
      returns with a = [4, 18, 21, 33, 17, 35, 20]
  mergeSort(a,0,3) returns with a as above
```

```
 mergeSort(a,0,6) calls mergeSort(a,4,6)
  mergeSort(a,4,6) calls mergeSort(a,4,5)
   mergeSort(a,4,5) calls mergeSort(a,4,4)
      mergeSort(a,4,4) returns
    mergeSort(a,4,5) calls mergeSort(a,5,5)
      mergeSort(a,5,5) returns
    mergeSort(a,4,5) calls merge(a,4,4,5)
      merge(a,4,4,5) merges [17] and [35] returns with a unchanged
    mergeSort(a,4,5) returns with a unchanged
  mergeSort(a,4,6) calls mergeSort(a,6,6)
    mergeSort(a,6,6) returns
  mergeSort(a,4,6) calls merge(a,4,5,6)
    merge(a,4,5,5) merges [17, 35] and [20]
      returns with a = [4, 18, 21, 33, 17, 20, 35]
  mergeSort(a,4,6) returns with a as above
mergeSort(a,0,6) calls merge(a,0,3,6)
  merge(a,0,3,6) merges [4, 18, 21, 33] and [17, 20, 35]
    returns with a = [4, 17, 18, 20, 21, 33, 35]
mergeSort(a,0,6) returns with a as above
```


Proof by Induction of Correctness and Termination for Merge Sort

Let's assume that `merge(a,i,m,j)` is correct and terminates, i.e. merges sorted sub-arrays `a[i..m]` and `a[m+1..j]` into a single sorted sub-array `a[i..j]`. This can be proven using loop invariants.

We prove correctness and termination of the `mergeSort(a,i,j)` method by induction on the size n of the sub-array considered, i.e. $n = \mathtt{j} - (\mathtt{i} - 1)$.

1) Base case $n = 1$: then `mergeSort(a,i,j)` immediately returns, and a subarray of size 1 is always sorted. So the method is correct.

2) For the recursive case: Assume that `mergeSort(a,i,j)` is correct and terminates for all sizes $n \leq k$ (induction hypothesis).

We must prove that that the method is correct for size $n = k + 1$.

Then `mergeSort(a,i,j)` calls `mergeSort(a,i,mid)` and `mergeSort(a,mid+1,j)`. In both cases, the size of the sub-array involved is $\leq k$, so by the induction hypothesis these two calls correctly sort the sub-arrays `a[i..mid]` and `a[mid+1..j]`.

After the recursive call, `merge(a,i,mid,j)` is called to merge the two sorted sub-arrays. By our assumption that `merge` is correct, this results in `a[i,j]` being sorted.

So `mergeSort` is correct for size $n = k + 1$.

Thus for all natural numbers n `mergeSort(a,i,j)` correctly sorts the sub-array `a[i..j]` where n is the size of the sub-array and terminates.

Anything that can be done using a loop can be done by recursion. E.g. finding student with smallest student number in an array:

```
public class Eg
{  public static final int MAX_STUDENTS = 150;

   static Student min(Student[] aClass, int aClassSize)
   {  return minBetween(aClass,0,aClassSize-1)
   }

   static Student minBetween(Student[] aClass, int from, int to)
   {  if (from > to) return null;
      else if (from == to) return aClass[from];
      else
      {  Student minRest =
            minBetween(aClass,from+1,to);
         assert(minRest != null);
         if (aClass[from].getNumber() < minRest.getNumber())
            return aClass[from];
         else
            return minRest;
      }
   }
   // main as before
}
```

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.

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.

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`.

But some algorithms can be coded much more simply using recursion and there is no loss of efficiency. One e.g., Merge Sort; to code it without using recursion, you must maintain your own stack of parts of the array that are put aside to be sorted later. Another e.g., traversing a tree and printing the labels on the nodes.

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" .

Analysis of Running Time of Merge Sort

Merging 2 arrays containing a total of n elements requires $n - 1$ comparisons in the worst case (when neither half runs out early). If you want to count all operations, then the running time will be $C_1 n + C_2$ where C_1 and C_2 are constants. If you drop the lower order term, you get $C_1 n$.

The running time for the complete sorting of an array of size n can be specified as a *recurrence relation:*

$$T(n) = \begin{cases} C_2 & \text{if } n \leq 1 \\ 2T(n/2) + C_1 n & \text{otherwise} \end{cases}$$

There are general methods for solving recurrence relations, but let's do it from first principles. If $n/2 > 1$, we can apply the recurrence relation to $T(n/2)$ to get:

$$\begin{aligned} T(n) &= 2(2T(n/4) + C_1 n/2) + C_1 n \\ &= 4T(n/4) + 2C_1 n \end{aligned}$$

Suppose that n is a power of 2, i.e. $n = 2^m$. Then we can keep applying the relation until n reaches $2^0 = 1$:

$$\begin{aligned} T(n) &= 2^m T(n/2^m) + mC_1 n \\ &= C_2 2^m + C_1 mn \text{ since } T(n/2^m) = T(1) = C_2 \\ &= C_2 n + C_1 n \log_2 n \text{ since } m = \log_2 n \\ &= O(n \log n) \end{aligned}$$

Thus, the running time of merge sort grows much more slowly than that of many other sorting algorithms whose running time is $O(n^2)$, e.g. selection sort, insertion sort, etc.:

n	$n \log_e n$	n^2
10	23	100
100	460	10,000
10^3	$6.9\ 10^3$	10^6
10^6	$13.8\ 10^6$	10^{12}