Induction and Recursion

Induction is a mathematical proof technique. When this technique is used in programming, it is called **recursion**. The difference is that induction applies to infinite sets and cannot be implemented, while recursion applies to finite sets and can be implemented. Induction/recursion is the fundamental mechanism for

- extending logical proof techniques into object domains and data structures,
- defining and building mathematical and programming objects,
- decomposing objects so that functions can be applied to the elementary units, and
- robust programming style and program verification.

Most practical computational problems are succinctly expressed in a recursive form (for instance, tree and graph traversal, spatial decomposition, divide-and-conquer algorithms, sorting, searching, and large classes of mathematical functions). As well, recursive function theory defines what can and cannot be computed.

Optimizing compilers usually convert recursion into do-loops. This is because the overhead of making multiple function calls during recursion is greater than a single do-loop. During the 1980s, programmers learned to write loops rather than recursions so that programs would run faster. This type of optimization is *irrelevant* today. It is far more important to enhance code readability, maintenance, and extendibility than to do an obsolete runtime optimization.

Recursive programming is mathematical programming. Once learned, almost all data structures and algorithms become both simpler and more comprehendible using recursion. The essential difficulty is that programmers need to be trained to think recursively.

Recursive Programming

Inductive definitions build up from the base case to any degree of complexity. Recursive programs reduce any degree of complexity one step at a time until the base case is reached. A recursion must be *well-founded*, that is, the steps must eventually terminate at the base. In most cases, the step increment is monotonically decreasing.

Recursive programs can be expressed in two forms, *mathematical* and *accumulating*. The mathematical form accumulates unevaluated operators on the outside and evaluates them after the base is reached. The accumulating form evaluates operators as they accumulate; when the base is reached, the result is returned. Accumulating functions have better performance than mathematical recursions, and easily compile into do-loops.

Mathematical:

```
if (base-case is true) then base-value else F[recursive-step]
```

Accumulating:

```
if (base-case is true) then accum else F[recursive-step, (accum + step)]
```

Mathematical Induction

Induction depends on a order relation over a domain `U`. The idea is to demonstrate truth for the base case (the simplest member of the ordered set), and then to demonstrate the truth for an arbitrary member of the set, assuming the truth of the member next to it in the order relation.

```
If N is an ordered set and property P isTrue for
      1) the minimal member of N, and
      2) if P(x) then P(next(x))
then P isTrue for all members x of N.
```

Using the natural numbers, `N = {1, 2,...}:`

```
If P(1) isTrue, and
      assuming P(x) we can show that P(x+1) isTrue, then
P(x) isTrue for all members of N.
```

Some Inductive Definitions

Base case: the value of *the most elementary case*

Examples:

`zero`	the additive identity
`one`	the multiplicative identity
`Phi`	the empty set
`nil`	the empty list, the empty tree
`false`	the logical ground

Generating rule: the transform which *defines the next case*, given an arbitrary case

Examples:

```
successor[n]      =     current[n] + 1
power-of-2[n]     =     2 * current[n]
summation[n]      =     n + current[n]
last[list]        =     rest[list] = nil
length[list]      =     length[rest[list]] + 1
member[x,S]       =     x=select[S] or member[x,rest[S]]
power-set[S]      =     current[S] * S
cardinality[S]    =     cardinality[rest[S]] + 1
node[btree]       =     left[btree] + right[btree]
logic-form[lf]    =     current[lf] implies next[lf]
parenthesis[pf]   =     "(" + current[pf] + ")" or
                              current[pf] + next[pf]
```

Recursion

Recursion is a form of induction in which we demonstrate two truths via computation. For the boolean property `P`,

```
1) P(minimal-member) isTrue

2)  If P(arbitrary-member) isTrue
      then P(arbitrary-member-one-step-closer-to-minimal-member) isTrue
```

Without algebraic processing, the second step requires an *instance* of the arbitrary member in order to perform computation. For example, say you are trying to find the product of `N` integers (the factorial function):

```
factorial[n] =def=

              (if n=1 then 1 else (n * factorial[n-1]))
```

This code requires a value of `N` in order to compute a value for the factorial of `N`. However, writing and proving the code itself requires mathematical induction. Above, the base case is when `N=1`. The *recursive invariant*, which is true for all `N>1`, is:

```
       factorial[n] = (n * factorial[n-1])
```

The recursive invariant is the general definition of the function. It states what remains the same when the value of `N` changes. When a function is written as a do-loop, the recursive invariant is called the *loop invariant*. Again, it defines what is always true each time through the loop. By verifying that a loop or a recursion maintains its invariant, it is possible to prove that code performs correctly.

The accumulating version of the factorial function is

```
factorial-acc[n, acc] =def=

                     (if n=1 then acc else factorial-acc[n-1, n*acc])
```

Recursive Function Exercises

Write recursive procedures (actual or pseudo-code) for the following functions. Where possible, write both the mathematical and the accumulating versions. Example:

```
+[i,j] =def= (if j=0 then i else (+[i,j-1] + 1))

+[i,j] =def= +acc[j,i]

    +acc[j,acc] =def= (if j=0 then acc else +acc[j-1,acc+1])
```

Integer Domain: `{i,j,k,n}` are positive integers. Note that the definitions below use mathematical induction; a recursive procedure must phrase the ordering relation in a descending form.

```
i+j    =def=  i+0 = i
              i+next[j] = (i+j) + 1

i*j    =def=  i*0 = 0
              i*next[j] = (i*j) + i

i^j    =def=  i^0 = 1
              i^next[j] = (i^j) * i

sum[n] =def=  sum[0] = 0
              sum[i+1] = sum[i] + (i+1)

fac[n] =def=  fac[0] = 1
              fac[i+1] = fac[i] * (i+1)

fib[n] =def=  fib[1] = fib[2] = 1
              fib[i+2] = fib[i+1] + fib[n]
```

Using *mathematical induction*, prove the following for integers:

```
(i*i) = (i^2)

(i*j) + (i*k) = i*(j+k)

(2*sum[n]) = n*(n+1)

(n^2) = (2*sum[n-1]) + n

(3*sum[n^2]) = (2*n + 1) * sum[n]

fib[n^2] = fib[n+1] * fib[n]

sum[n^3] = (sum[n]^2)
```

List Domain: `{x,y}` are elements, `list` is a list. Write these recursive functions:

`last[list]`

returns the last element of `list`.

`length[list]`

returns the length of `list`.

`member[x,list]`

Boolean, returns `T` iff `x` is a member of `list`.

`copy[list]`

returns a copy of `list`.

`alternating-elements[list]`

returns a new list of every other element of `list`.

`reverse[list]`

returns a new list with the elements of `list` in reverse order.

`samelength[list1,list2]`

Boolean, returns `T` iff both lists have the same length
(Do not use any integer arithmetic for this.)

`intersection[list1,list2]`

returns a new list which contains elements in both `list1` and `list2`.

`list-equal[list1,list2]`

Boolean, returns `T` iff `list1=list2` (elements in same order).

`set-equal[list1,list2]`

Boolean, returns `T` iff both `list1` and `list2` have the same elements,
not necessarily in order.

`append[list1,list2]`

returns a new list which is `list1` appended to `list2`.

`substitute[x,y,list]`

returns a new list with `x` substituted for every occurrence of `y` in `list`.

Using *mathematical induction*, prove the following properties about list functions (harder):

```
length[append[list1,list2]] = length[list1] + length[list2]

last[list] = first[reverse[list]]

samelength[list, reverse[list]]

reverse[reverse[list]] = list

substitute[x,y,substitute[y,x,list]] =/= list
```

Tree Domain: `tree` is an arbitrary tree.

`depth[tree]`

returns the maximum depth of `tree`.

`flatten[tree]`

returns a list of all nodes of `tree`

`fringe[tree]`

returns a list of the leaf nodes of `tree`

Function Domain: `f` is a function; `p` is a Boolean function

`andlist[list]`

returns the Boolean AND of a list of Boolean elements

`map[f,list]`

returns a new list with the function `f` applied to each element of `list`

`mapchoose[p,list]`

returns a new list with every element from `list` which satisfies test `p`

Challenge: Implement the following recursive functions over integers and trace their execution. What do you observe? (Try `fm` for `i= {341,96,104,336,133}`; try `ackermann` and `takeuchi` for small integers only.)

```
fm[i] =def=

            if i=1 then stop else
                  if even[i] then fm[i/2] else
                        fm[(3i+1)/2]

ackermann[i,j] =def=

            if i=0 then j+1 else
                  if j=0 then ackermann[i-1,j] else
                        ackermann[i,j-1]

ackermann2[i,j] =def=

            if i=0 then j+1 else
                  if j=0 then ackermann[i-1,1] else
                              ackermann[i-1,ackermann[i,j-1]]
```

```
takeuchi[i,j,k] =def=

  if (i =< j) then k else
    takeuchi[takeuchi[i-1,j,k],takeuchi[j-1,k,i],takeuchi[k-1,i,j]]
```