Introduction to Recursion

- A subroutine/function is called *recursive* if it calls itself.
- If a subroutine/function simply called itself as a part of its execution, it would result in infinite recursion. This is a bad thing.
- Therefore, when using recursion, one must ensure that at some point, the subroutine/function terminates without calling itself.
- **Example:** The factorial function $n!$ can be implemented recursively.

```
int factorial(int n) {
   if (n<=1)
      return 1;
   else
      return n*factorial(n-1);
}
```

- What ensures that the function *factorial* terminates?

Where is Recursion Seen/Used?

- We occasionally see recursion in the "real" world:
  - Russian Matryoshka (nested dolls)
  - Two almost parallel mirrors
  - A video camera pointed at the monitor
- More importantly for us, it is useful for some data structures and the associated algorithms.
- Some problems can be solved by combining solutions of smaller instances of the given problem. Recursion can be useful in these cases.
- **Examples:**
  - Binary Search
  - Mergesort
  - Computing $n!$
  - Many *divide-and-conquer* algorithms

Recursion Example

- Suppose we want to implement a subroutine **CountDown**(n) which outputs the integers from n down to 1, where $n > 0$.
- **Example:** The call **CountDown(5)** results in output '5 4 3 2 1'.
- The best way to implement this is a loop:

```
void CountDown(int n) {
        for (i=n;i>0;i--)
           cout<<i<<" ";
        cout<<"\n";
        }
```

- Of course, if we did this, we wouldn't learn anything about recursion. So, let's consider how to do it with recursion.

Recursive CountDown(n)

- How can we think of this subroutine recursively?
- **CountDown(n)** outputs n followed by the numbers from $n - 1$ down to 1.
- The numbers $n - 1$ down to 1 are the output from **CountDown($n - 1$)**.
- Thus, the output from **CountDown(n)** is n followed by the output from **CountDown($n - 1$)**.
- Thus, we can write the function recursively as follows:

```
void CountDown(int n) {
          cout<<n<<" ";
          CountDown(n-1):
          }
```

- Nice, but something is wrong here. What is it?

Recursive CountDown(n): Error

- The problem is, CountDown never stops:

Execute	Output	Then Execute
CountDown(3)	3	**CountDown(2)**
CountDown(2)	2	**CountDown(1)**
CountDown(1)	1	**CountDown(0)**
CountDown(0)	0	**CountDown(-1)**
CountDown(-1)	-1	**CountDown(-2)**
$\vdots$	$\vdots$	$\vdots$

- The problem is not with the recursion, but with our logic. We are supposed to stop printing when $n = 1$, but we didn't take that into account.
- To fix this, we modify it so that a call to **CountDown(0)** produces no output and does not call **CountDown** again.
- Calls to **CountDown(n)** when $n < 0$ should produce no output, either.
- We can take care of both of these problems at once.

Recursive CountDown(n): Fixed

- The following version of **CountDown(**n**)** is correct:

```
void CountDown(int n) {
       if(n>0) {
           cout<<n<<" ";
           CountDown(n-1):
           }
        }
```

- Now, **CountDown(**n**)** does exactly what we want when $n > 0$.
- It is not too difficult to see that if $n \leq 0$, the subroutine **CountDown(**n**)** does nothing.

Making Recursion Work

- In order for a recursive routine to work properly, it must be defined so that it will terminate eventually.
- Thus, a proper recursive definition has both of the following:
  - *base case(s)*: A case which is solved non-recursively. In other words, when a routine gets to the base case, it does not call itself again. This is also called a *stopping case* or *terminating condition*.
  - *inductive case(s)*: A recursive rule for all cases except the base case. An inductive case should always progress toward the base case.
- **Example:** For the routine **CountDown(n)**:
  - *base case*: When $n \leq 0$, **CountDown(n)** does nothing.
  - *inductive case*: When $n > 0$, **CountDown(n)** outputs n, and executes **CountDown($n - 1$)**. Notice, the second call is closer to the base case.

Example: Factorial

- Recursive definition:

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

- **Example:**

$$\begin{aligned} 1! &= 1 \\ 2! &= 2 \times (1)! = 2 \times 1 = 2 \\ 3! &= 3 \times (2)! = 3 \times 2 = 6 \\ 4! &= 4 \times (3)! = 4 \times 6 = 24 \end{aligned}$$

- In general, when $n > 1$,

$$n! = n * (n-1) * (n-2) * (n-3) * ... * 1$$

- In C++, we can implement this as:

```
int factorial(int n) {
   if(n==1)
      return 1;
   else
      return n*factorial(n-1);
   }
```

Recursive Problem Solving

In general, we can solve a problem with recursion if we can:

- Find one or more simple cases of the problem that can be solved directly.
- Find a way to break up the problem into smaller instances of the same problem.
- Find a way to combine the smaller solutions.

Recursion and Memory

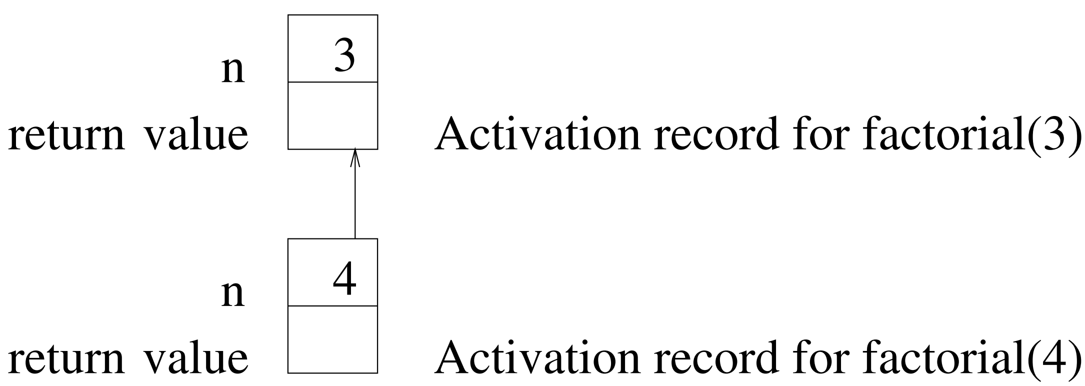
- Each call of a function generates an instance of that function
- An instance of a function contains
  - memory for each parameter (input)
  - memory for each local variable
  - memory for the return value

  This chunk of memory is referred to as an *activation record*.
- Thus, a recursive function that calls itself n times must allocate n activation records.
- Usually, an iterative implementation will require on the order of one activation record, plus a constant amount of space.
- This is the reason recursion is avoided when possible. In fact, good compilers remove recursion whenever possible.

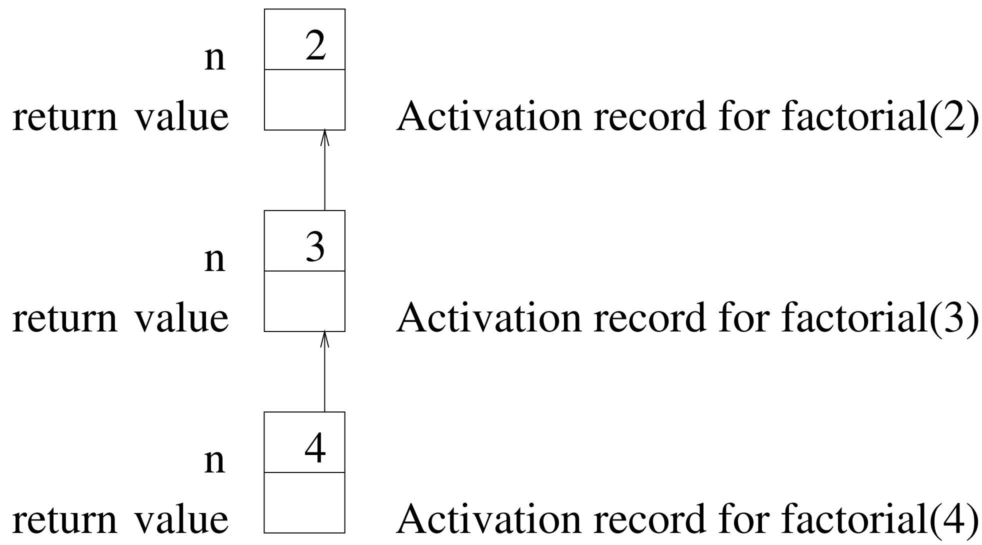
Example:

- **factorial(4)**

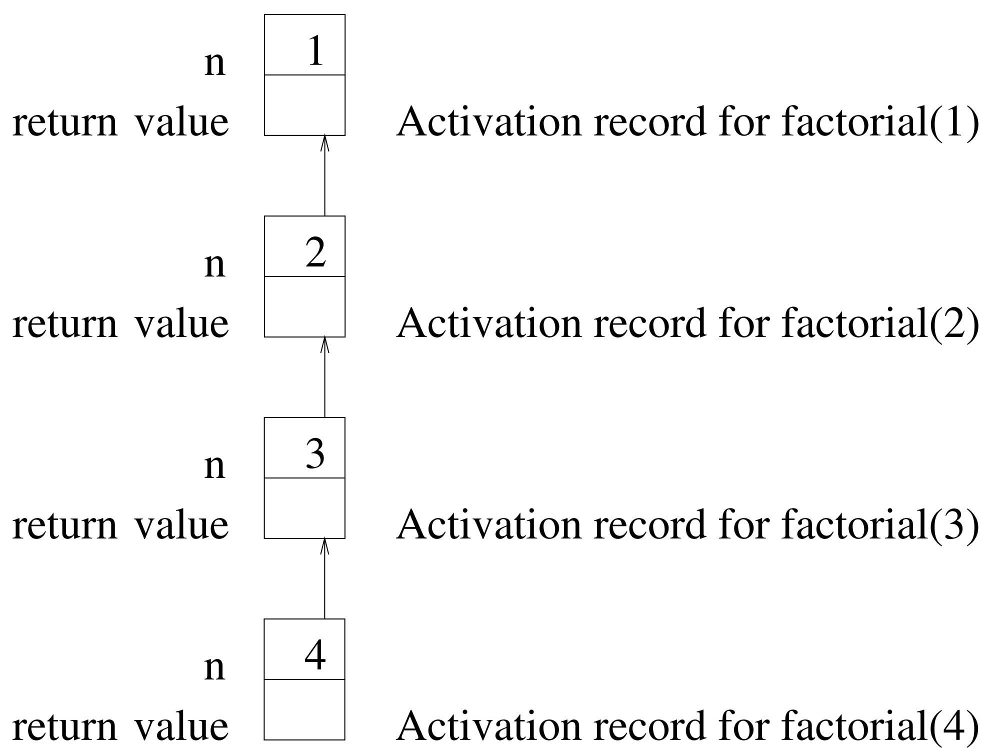
- **factorial(4)**: call to factorial(3)



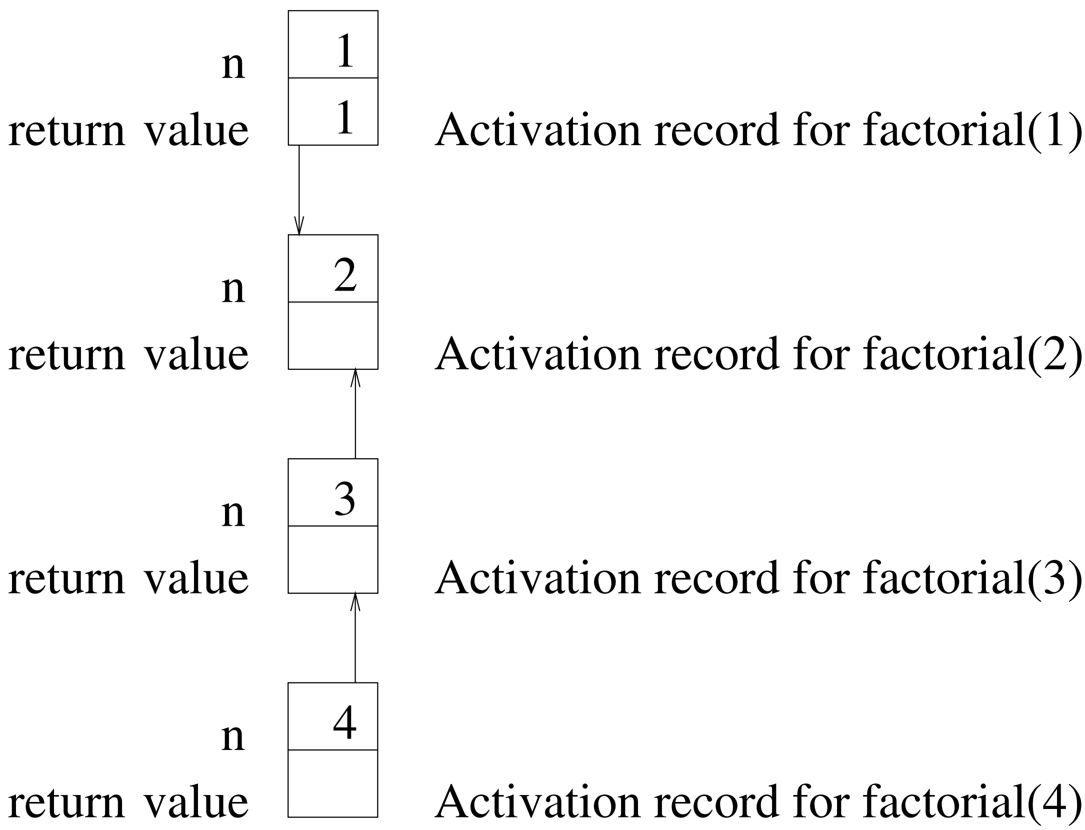
- **factorial(4):** call to factorial(2)



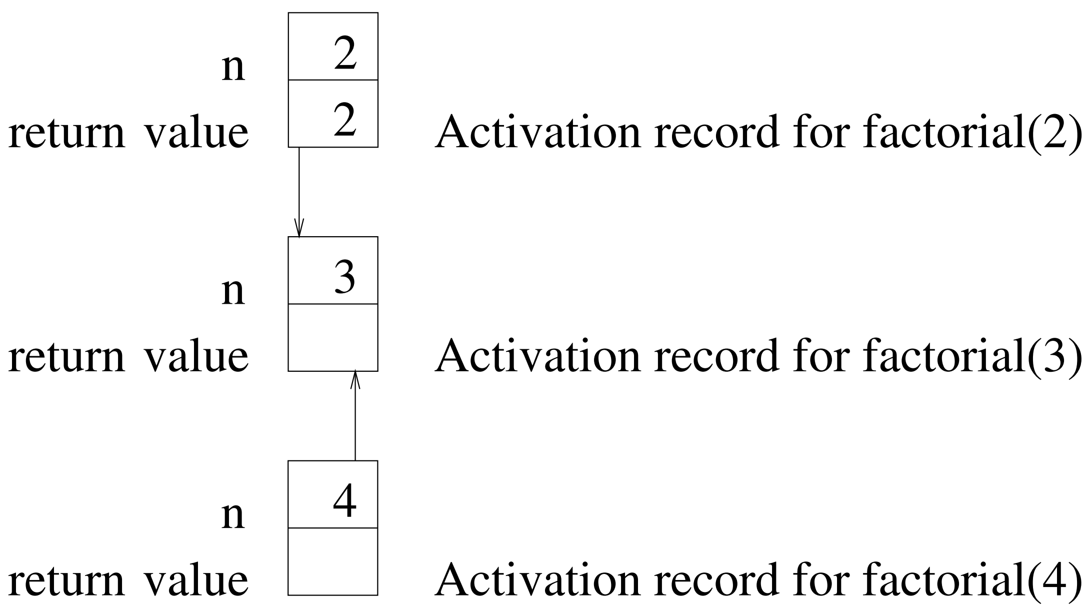
- **factorial(4):** call to factorial(1)



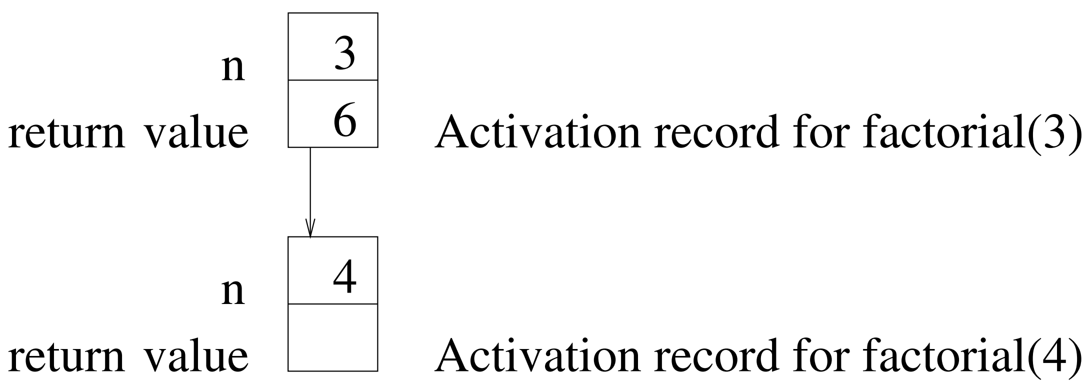
- **factorial(4):** factorial(1) is the base case, so it returns ‘1’.



- **factorial(4):** factorial(2) now returns '2'.



- **factorial(4):** factorial(3) now returns ‘6’.



- **factorial(4):** returns ‘24’. This was the original function call, so the execution is finished.

The Run-Time Stack

- In order to support recursive function calls, the run-time system treats memory as a stack of activation records
- Computing **factorial(n)** recursively requires the allocation of n activation records on the stack.
- What if we have infinite recursion:

```
int infiniteRecursion(int n) {
   if (n==0)  return 1;
   else       return infiniteRecursion(n);
}
```

  The value of n never reaches zero, so the function is called, and records are pushed onto the stack, until the system runs out of memory.
- Even if our recursion is not infinite, it is possible that the recursion runs too deep, since computers only have a finite amount of memory.

Recursion and Iteration

Recursive functions can be translated to functions that use loops.

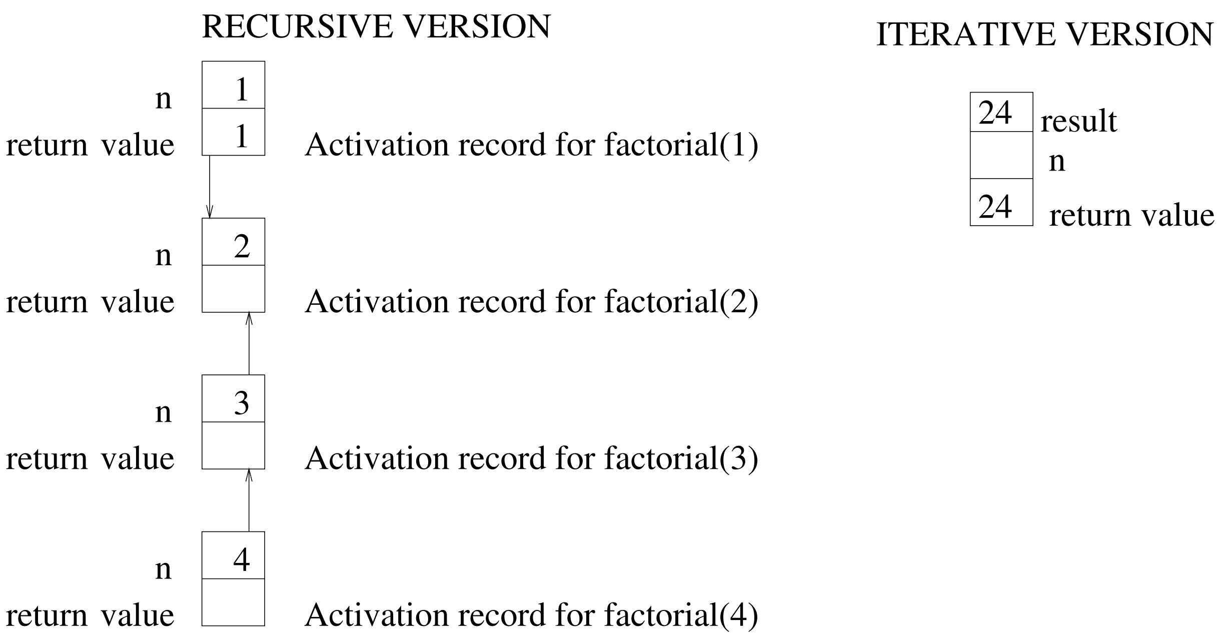
- **Recursive:**

```
int factorial(int n) {
   if(n==1)
      return 1;
   else
      return n*factorial(n-1);
}
```

- **Iterative:**

```
int factorial (int n) {
   int result=1;
       while (n>1) {
          result = result * n;
          n--;
       }
       return result;
}
```

Memory Usage?



Notice that for input n, the recursive implementation needs to allocate $2n$ integers, while the iterative implementation needs only 3.

Recursion: Advantages

- Recursion often mimics the way we think about a problem, thus the recursive solutions can be very intuitive to program. For example, binary search is very similar to the way we search through the phone book.
- Often recursive routines to solve problems can be much shorter than iterative (non-recursive) routines. This can make the code easier to understand, modify, and/or debug.
- Many of the ‘best’ known algorithms for many problems are based on a divide-and-conquer approach:
  - Divide the problem into a set of smaller problems
  - Solve each small problem separately
  - Put the results back together for the overall solution
- These divide-and-conquer techniques are often best thought of in terms of recursive functions. (e.g. Quicksort and Mergesort)

Recursion: Disadvantages

- Each time one subroutine calls another, the computer's operating system must take care of a number of things:
  - Recording how to re-start the calling subroutine later on,
  - Passing the parameters from the calling subroutine to the called subroutine (often by pushing the parameters onto a stack controlled by the system)
  - Setting up space for the called subroutine's local variables
  - Recording where the calling subroutine's local variables are stored
- Doing all this requires time and memory.
- Thus a routine which makes many recursive calls can require a lot of time and memory - more than a non-recursive solution might.

Common Recursion Errors

- Forgetting or having incomplete base cases.
- **Example:** This routine goes into infinite recursion if given a negative number for N.

```
void Sum1toN(int N)
{
   if (N == 0) return(0);
   else return(N + Sum1toN(N-1));
}
```

- Getting things backwards.
- **Example:** One of these routines prints from 1 up to N, the other from N down to 1. Which is which?

```
     void PrintN(int N) {
          if (N > 0) {
             PrintN(N-1);
             cout << N << ", ";
             }
          }
//---------------------------
     void NPrint(int N) {
        if (N > 0) {
           cout << N << ", ";
           NPrint(N-1);
           }
        }
```

Example 2: Fibonacci Numbers

- The Fibonacci numbers are a sequence of integers which are of interest in mathematical and computing applications
- They are given by:

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

- Thus, the first few are:

```
Fib(0) = 0
Fib(1) = 1
Fib(2) = Fib(0) + Fib(1) = 0 + 1 =  1
Fib(3) = Fib(1) + Fib(2) = 1 + 1 =  2
Fib(4) = Fib(2) + Fib(3) = 1 + 2 =  3
Fib(5) = Fib(3) + Fib(4) = 2 + 3 =  5
Fib(6) = Fib(4) + Fib(5) = 3 + 5 =  8
Fib(7) = Fib(5) + Fib(6) = 5 + 8 = 13
etc.
```

- We will consider one iterative solution and one recursive solution to calculate Fib(N)

Iterative Fibonacci Routine

- We can calculate the Fibonacci numbers iteratively by starting with Fib(0)=0 and Fib(1)=1, and and then working forward until we reach Fib(N).
- Since Fib(x) = Fib(x-1) + Fib(x-2), we must keep track of the previous two numbers as we go.

```
int Fib(int N) {
    int fib, fibm1, fibm2, index;
    if (N <= 1) return(N);
    else {
        fibm2 = 0;
        fibm1 = 1;
        index  = 1;
        while (index < N) {
            fib = fibm1 + fibm2;
            fibm2 = fibm1;
            fibm1 = fib;
            index = index + 1;
            }
        return(fib);
       }}
```

Recursive Fibonacci Solution

- While the iterative solution started from Fib(0) and Fib(1) and worked forward, the recursive solution starts from N and works backward.
- We use Fib(N) = Fib(N-1) + Fib(N-2) as before.

```
int Fib(int N) {
   if (N <= 1)
      return(N);
   else
      return(Fib(N-1) + Fib(N-2));
   }
```

- As you can see, the recursive routine is much simpler to program.
- If you try both programs for assorted values of N, however, you will also see that the iterative routine is much more efficient.

Recursion: Conclusion (1)

- A recursive function is one that invokes another instance of itself.
- Recursion is an alternative to iteration.
- Recursion can often provide a more elegant solution than iteration.
- Each instance of a function has its own set of local variables and parameters.
- Recursive solutions are often less efficient, in terms of time and space, than an iterative solution.
- Some data structure problems are difficult to solve without recursion. (particularly when the data structure is recursive in its definition).

Recursion: Conclusion (2)

- Recursion can be used when all of the following conditions can be satisfied:
  - There exists one or more simple solutions to the problem.
  - Other cases of the problem can be expressed in terms of one or more reduced cases of the problem (which are closer to the known simple solutions).
  - Eventually the problem can be reduced to one of the simple solutions.
- When designing a recursive algorithm it is important to ensure that the recursion will eventually reach a terminating condition and stop.