Chapter 17 - Notes
Recursion

I. Recursive Definitions

A. **Recursion**: The process of solving a problem by reducing it to smaller versions of itself.

B. **Recursive Function**: A function that calls itself.

C. **Recursive Definition**: A definition in which something is defined in terms of a smaller version of itself.

1. A valid Recursive Definition requires 3 properties:
   a. Every recursive definition must have one or more base cases.
   b. The general case must eventually be reduced to the base case.
   c. The base case stops the recursion.

D. **Base Case**: The case for which the solution is obtained directly.

E. **Example of Recursion Using Factorials**

1. In Mathematics, the factorial of an integer is defined as follows:

   \[ n! = 1 \quad \text{if} \quad n = 0 \]

   \[ n! = n \times (n - 1)! \quad \text{if} \quad n > 0 \]

2. Using our definitions from above, \( 0! = 1 \) would be our **base case**, since the solution is obtained directly.

3. The **general case** would be \( n! = n \times (n - 1)! \). The solution to this part of the algorithm must be obtained by first solving the case of \( (n - 1)! \).

4. Now let's look at the problem of finding a solution for \( 3! \).
   a. First notice that per definition, \( 3! \) is defined as the following:

   \[ 3! = 3 \times 2! \]
b. Before we can solve this problem we must first determine the solution of $2!$, which is defined by our general definition as the following:

$$2! = 2 \times 1!$$

By using substitution, we obtain the following formula:

$$3! = 3 \times (2 \times (1!))$$

But we still have to solve for $1!$.

c. The definition for $1!$ is as follows (our general definition):

$$1! = 1 \times 0!$$

Again we substitute:

$$3! = 3 \times (2 \times (1 \times (0!)))$$

We must now solve for $0!$ before we can determine a solution.

d. At this point, $n$ is not greater than 0 (zero), therefore:

$$0! = 1$$

Now we can make the final substitution that will allow us to compute the problem.

$$3! = 3 \times (2 \times (1 \times (1)))$$

Hence: $3! = 6$

F. **The Factorial Problem using C++ Code**

```cpp
1. int fact ( int num )
   {
      if( num == 0 )      \< Base Case
         return 1 ;
      else
         return num * fact ( num - 1 ) ; \< General Case
   }
```

-2-
2. Think of a recursive function as having unlimited copies of itself that can be called.

3. Every call to a recursive function - that is, every recursive call - has its own code and its own set of parameters and local variables.

4. After a function completes its execution, it returns a value and control back to the environment (calling function) which was the previous call.
   a. Notice that the current call must execute completely before control goes back to the previous call.
   b. The execution in the previous call begins from the point immediately following the recursive call.

G. **Direct and Indirect Recursion**
   1. **Direct Recursion**: A function is called "direct" if it calls itself.
   2. **Indirect Recursion**: A function that calls another function that eventually calls the original calling function is said to be "indirect" recursion.
      a. Example: If function A calls function B, which in turn calls function A, then function A is said to be indirectly recursive.
      b. Example: If function A calls function B, which calls function C, which calls function D, which then calls function A, then function A is said to be indirectly recursive.
   3. **Tail Recursive Function**: A recursive function in which the last statement executed is the recursive call. The above example function `fact` is an example of a tail recursive function.

H. **Infinite Recursion**
   1. Like loops, recursively called functions can result in an infinite loop. In the case of loops, the object is to get closer and closer to the case that will prove false (the exit condition) and throw the program out of the loop.
   2. With recursive functions, the object is to get closer to the base case.
3. If every recursive call results in another recursive call, then the recursive function is said to have infinite recursion. *(The dreaded infinite recursion!!)*

4. The goal is to have a sequence of recursive calls eventually reach a base case.

5. It is important to design a recursive function with the base case in mind and how it will eventually reach it.

II Problem Solving Using Recursion

A. **Example Problem:** Finding the largest element in a list using recursion rather than loops.

1. *Suppose we have a list inside of an array such as the following:*

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>myList</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
<td>10</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

2. *Now Let's think in terms of recursion:*

   a. If the size of our list is greater than one, then we can compare `myList[0]` with the balance of the list ( `myList[1]...myList[5]` ) to see which has the higher number.

   b. Before we can do the above, we must determine what the highest number in `myList[1]...myList[5]` is. Let's break this down into the same problem:


   c. As you can readily see, we need to continue to break the problem down the same way until we get to a list size of one.


      `myList[5]`
d. It is trivial to determine which value is the largest in a list of size one.

    return the value in myList[5]

e. Now compare the value in myList[4] to the value in myList[5] and return the larger of the two. (In the case of our example, it would be myList[4]).

f. Now myList[3] gets compared to the largest value of myList[4]...myList[5] (which turns out to be myList[4]) and returns the larger of the two values (which turns out to be myList[3]).

g. Eventually, we will reach a solution when myList[0] is compared to the largest value of myList[1]...myList[5] which is 10.

3. **Recursive Function as Written in C++**

```cpp
int largest ( const int myList [], int lowIndex, int upperIndex )
{
    int max;

    if ( lowIndex == upperIndex ) ← Base Case
        return myList[lowIndex];
    else
    {
        max = largest ( myList , lowIndex + 1 , upperIndex ) ;
        if ( myList[lowIndex] >= max )
            return myList[lowIndex];
        else
            return max;
    }
}
```

-5-
4. Same Recursive Function Used in a Program

```cpp
#include <iostream>
using namespace std;

int largest( const int list[] , int lowIndex , int upperIndex ) ;

int main ( )
{
    int myList [10] = { 23, 43, 35, 38, 67, 12, 76, 10, 34, 8 } ;

    cout << "The largest element in myList is: " << largest( myList , 0 , 9 ) ;
    cout << endl << endl ;
    system("pause") ;
    return 0 ;
}

int largest( const int myList[] , int lowIndex , int upperIndex )
{
    int max;

    if ( lowIndex == upperIndex )
        return myList[lowIndex] ;
    else
    {
        max = largest( myList , lowIndex + 1 , upperIndex ) ;
        if ( myList[lowIndex] >= max )
            return myList[lowIndex] ;
        else
            return max ;
    }
}
```