Recursion

• Recursion is a fundamental programming technique that can provide an elegant solution certain kinds of problems
Recursive Thinking

- A recursive definition is one which uses the word or concept being defined in the definition itself.

- When defining an English word, a recursive definition is often not helpful.

- But in other situations, a recursive definition can be an appropriate way to express a concept.

- Before applying recursion to programming, it is best to practice thinking recursively.
Recursive Definitions

• Consider the following list of numbers:

   24, 88, 40, 37

• Such a list can be defined as follows:

   A LIST is a: number
   or a: number comma LIST

• That is, a LIST is defined to be a single number, or a number followed by a comma followed by a LIST

• The concept of a LIST is used to define itself
Recursive Definitions

- The recursive part of the LIST definition is used several times, terminating with the non-recursive part:

```
number comma LIST
24 , 88, 40, 37
```
Infinite Recursion

- All recursive definitions have to have a non-recursive part
- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause *infinite recursion*
- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
- The non-recursive part is often called the *base case*
Recursive Definitions

• N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive.

• This definition can be expressed recursively as:

\[ 1! = 1 \]
\[ N! = N \times (N-1)! \]

• A factorial is defined in terms of another factorial.

• Eventually, the base case of 1! is reached.
Recursive Definitions

\[5! = 5 \times 4! = 4 \times 3! = 3 \times 2! = 2 \times 1! = 1\]

Values:
- \(120\) for \(5!\)
- \(24\) for \(4!\)
- \(6\) for \(3!\)
- \(2\) for \(2!\)
- \(1\) for \(1!\)
Recursive Programming

• A Function can invoke itself; if set up that way, it is called a recursive function

• The code of a recursive function must be structured to handle both the base case and the recursive case

• As with any function call, when the function completes, control returns to the function that invoked it (which may be an earlier invocation of itself)
Recursive Programming

- Consider the problem of computing the sum of all the numbers between 1 and any positive integer \( N \).
- This problem can be recursively defined as:

\[
\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + (N-1) + \sum_{i=1}^{N-2} i \\
= N + (N-1) + (N-2) + \sum_{i=1}^{N-3} i \\
\vdots
\]

\[
= N + (N-1) + (N-2) + \cdots + 1
\]
Recursive Programming

// This function returns the sum of 1 to num
int sum (int num)
{
    int result;

    if (num == 1)
        result = 1;
    else
        result = num + sum (n-1);

    return result;
}
Recursive Programming

main

sum(3)

result = 6

sum

result = 3

sum(2)

result = 1

sum(1)

result = 1

sum

result = 1
Recursive Programming

- Note that just because we can use recursion to solve a problem, doesn't mean we should.

- For instance, we usually would not use recursion to solve the sum of 1 to N problem, because the iterative version is easier to understand.

- However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version.

- You must carefully decide whether recursion is the correct technique for any problem.
Indirect Recursion

- A function invoking itself is considered to be *direct recursion*

- A function could invoke another function, which invokes another, etc., until eventually the original function is invoked again

- For example, function $m_1$ could invoke $m_2$, which invokes $m_3$, which in turn invokes $m_1$ again

- This is called *indirect recursion*, and requires all the same care as direct recursion

- It is often more difficult to trace and debug
Indirect Recursion
Towers of Hanoi

• The *Towers of Hanoi* is a puzzle made up of three vertical pegs and several disks that slide on the pegs.

• The disks are of varying size, initially placed on one peg with the largest disk on the bottom with increasingly smaller ones on top.

• The goal is to move all of the disks from one peg to another under the following rules:
  - We can move only one disk at a time.
  - We cannot move a larger disk on top of a smaller one.
Towers of Hanoi

Original Configuration

Move 1

Move 2

Move 3
Towers of Hanoi

Move 4

Move 5

Move 6

Move 7 (done)
Towers of Hanoi

• An iterative solution to the Towers of Hanoi is quite complex

• A recursive solution is much shorter and more elegant
Towers of Hanoi

```c
#include <stdio.h>
#include <conio.h>

void transfer(int,char,char,char);

int main()
{
    int n;
    printf("Recursive Solution to Towe of Hanoi Problem\n");
    printf("enter the number of Disks\n");
    scanf("%d",&n);
    transfer(n,'L','R','C');
    getch();
    return 0;
}

void transfer(int n,char from,char to,char temp)
{
    if (n>0)
    {
        transfer(n-1,from,temp,to);  /* Move n-1 disk from origin to temporary */
        printf("Move Disk %d from %c to %c\n",n,from,to);
        transfer(n-1,temp,to,from);  /* Move n-1 disk from temporary to origin */
    }
    return;
}
```
Drawbacks of Recursion

Regardless of the algorithm used, recursion has two important drawbacks:

- Function-Call Overhead
- Memory-Management Issues
A special kind of recursion is tail recursion.

- *Tail recursion* is when a recursive call is the last thing a function does.

Tail recursion is important because it makes the recursion $\rightarrow$ iteration conversion very easy.

- That is, we like tail recursion because it is easy to eliminate.
- In fact, tail recursion is such an obvious thing to optimize that some compilers automatically convert it to iteration.
Eliminating Recursion — Tail Recursion

For a void function, tail recursion looks like this:

```c
void foo(TTT a, UUU b)
{
    ...
    foo(x, y);
}
```

For a function returning a value, tail recursion looks like this:

```c
SSS bar(TTT a, UUU b)
{
    ...
    return bar(x, y);
}
```
A tail-recursive Factorial Function

We will use an auxiliary function to rewrite factorial as tail-recursive:

```c
int factAux (int x, int result)
{
    if (x==0) return result;
    return factAux(x-1, result * x);
}

int tailRecursiveFact( int x)
{
    return factAux (x, 1);
}
```