Iteration

Looping solution through successive (sub)cases of a problem

Example: Factorials

$$N! = N * (N-1) * (N-2) * \ldots * 2 * 1 \quad (N > 0)$$

```
int IterativeFact(const int N)
// ----------------------------------------------------
// Computes the factorial of a nonnegative integer.
// Precondition: N must be greater than or equal to 0.
// Postcondition: Returns the factorial of N; N is
//       unchanged.
// ----------------------------------------------------
{
   int prod = 1;

   for(int i = N; i > 1; --i)      // iterate downwards
   {
      prod = prod * i;             // or  prod *= i;
   }

   return(prod);
}
```

Observations:

- Requires temporary variable (prod)
- Requires looping mechanism: loop index counter initialization, completion testing, incrementing
- Loop boundary conditions -- critical

Recursion

Define problem solution in terms of smaller problems of the same type.

Example: Factorials

```
N! = N * (N-1) * (N-2) * . . . * 2 * 1
      (N > 0)
```

$$N! = N * \underbrace{\boxed{(N-1) * (N-2) * . . . * 2 * 1}}_{(N-1)!}$$

```
N! = N * (N-1)!                    (N > 0)
0! = 1                             (N = 0)  BASE CASE
```

```
int Fact(const int N)
// ------------------------------------------------
// Computes the factorial of a nonnegative integer.
// Precondition: N must be greater than or equal to 0.
// Postcondition: Returns the factorial of N; N is
//      unchanged.
// ------------------------------------------------
{
   if (N == 0)                        // Base Case
      return 1;

   else
      return N * Fact(N-1);           // Recursive call

}
```

Characteristics of a recursive function:

- **A recursive function calls itself (one or more times)**
- **Each recursive call solves an identical, but smaller, problem**
- **A test for the base case enables the recursive calls to stop**
- **Eventually, one of the smaller problems must be the base case**

Key Concepts for Constructing Recursive Solutions:

1. **How can you define the problem in terms of a smaller problem of the same type?**
2. **How does each recursive call diminish the size of the problem?**
3. **What instance of the problem can serve as the base case?**
4. **As the problem size diminishes, will you reach this base case?**

Advantage of Recursion:

- **Often "elegant" or "simple" solution**

DisAdvantage of Recursion:

- **None in general**
- **Sometimes execution "efficiency" (vs. "iterative" solution)**
- **May be "harder" to debug ("box" drawings)**

Example: Recursive void function to write a String backwards

H	i		T	h	e	r	e		

```
const int MAX_LENGTH = 30;
typedef char stringType[MAX_LENGTH+1];

void WriteBackward(stringType S, int Size)
// ---------------------------------------------------
// Writes a character string backward.
// Precondition: The string S contains Size
//       characters, where Size >= 0.
// Postcondition: S is written backward, but remains
//       unchanged.
// ---------------------------------------------------
{
   if (Size == 0)              // base case - do nothing
      return;

   cout << S[Size-1];          // write last character

   WriteBackward(S, Size-1);   // rest of string

}
```

(tail recursion)

```
// Iterative version.
void WriteBackward(stringType S, int Size)
{
   while (Size > 0)
   {
      cout << S[Size-1];

      --Size;
   }
}
```

Example: Compute terms in Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

```
Fib(N) = Fib(N-2) + Fib(N-1)
       =  1                    N = 1 or 2

          (2 base cases)
```

```
int Rabbit(int N)
// ----------------------------------------------------
// Computes a term in the Fibonacci sequence.
// Precondition: N is a positive integer.
// Postcondition: Returns the Nth Fibonacci number.
// ----------------------------------------------------
{
   if (N <= 2)
      return 1;

   else
      return Rabbit(N-1) + Rabbit(N-2);
}
```

```
int RabbitIterative(int N)
{
     if (N <= 2)
          return 1;

     // Iterate:   ... rab_2  rab_1  rab ....
     int rab_2 = 1;          // 1
     int rab_1 = 1;          // 2
     int rab;

     for(int i = 3; i <= N; ++i)
     {
          rab = rab_2 + rab_1;     // Sum of prior two iterations
          rab_2 = rab_1;
          rab_1 = rab;
     }

     return(rab);
}
```

Example: Determine how many ways to choose K items from a collection of N items.

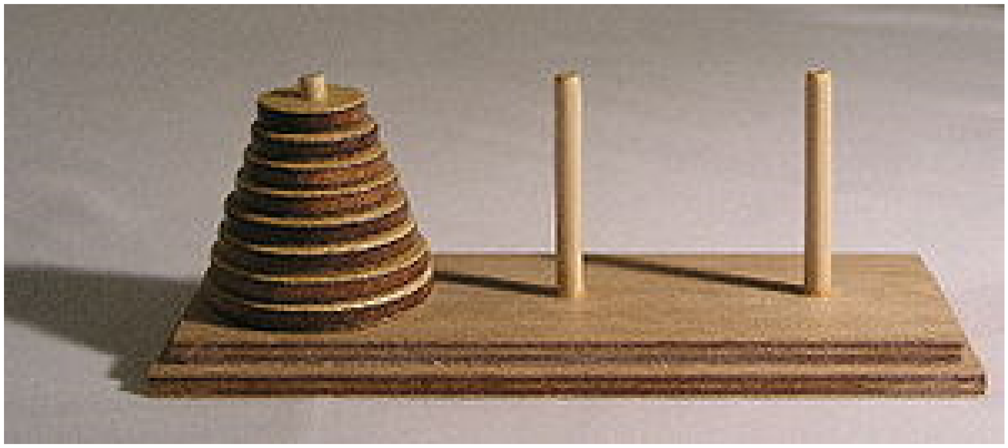
```
int Choose(int N, int K)
// ----------------------------------------------------
// Computes the number of groups of K out of N things.
// Precondition: N and K are nonnegative integers.
// Postcondition: Returns Choose(N, K).
// ----------------------------------------------------
{
   if ( (K == 0) || (K == N) )
      return 1;

   else if (K > N)
      return 0;

   else
      return( Choose(N-1, K-1) + Choose(N-1, K) );
}
```

Example: Searching Lists

- **Find largest item in arbitrary list**
- **Find kth smallest item in a list**
- **Find item in sorted list**

```
void BinarySearch(const int A[], int First, int Last,
                  int Value, int& Index)
// ---------------------------------------------------
// Searches the array items A[First] through A[Last]
//       for Value by using a binary search.
// Precondition: 0 <= First, Last <= SIZE-1, where
//       SIZE is the maximum size of the array, and
//       A[First] <= A[First+1] <= ... <= A[Last].
// Postcondition: If Value is in the array, Index is
//       the index of the array item that equals Value;
//       otherwise Index is -1.
// -----------------------------------------------------
{
   if (First > Last)
      Index = -1;      // Value not in original array

   else
   {
      // Invariant: If Value is in A,
      //   A[First] <= Value <= A[Last]

      int Mid = (First + Last)/2;

      if (Value == A[Mid])
         Index = Mid;  // Value found at A[Mid]

      else if (Value < A[Mid])
         BinarySearch(A, First, Mid-1, Value, Index);

      else
         BinarySearch(A, Mid+1, Last,  Value, Index);
   }
}
```

Example: Towers of Hanoi



Goal: Move initial stack of disks to another pole.
Rules:
1) Move 1 disk at a time.
2) Never place a larger disk on top of a smaller disk.

```
// Recursive solution is much more obvious than iterative
// How would you solve this problem iteratively???
```

```
void SolveTowers(int Count,
                 char Source,
                 char Destination,
                 char Spare)
{
   if (Count == 1)
   {
      cout << "Move top disk from pole " << Source
           << " to pole " << Destination << endl;
   }
   else
   {
      SolveTowers(Count-1, Source, Spare, Destination);

      SolveTowers(1,       Source, Destination, Spare);

      SolveTowers(Count-1, Spare, Destination, Source);
   }
}
```