Recursive Algorithms

Recursion

- The *factorial of N* is the product of the first N positive integers:

$$N * (N - 1) * (N - 2) * \cdots * 2 * 1$$

- The factorial of N can be defined *recursively* as

```
                    ⎧  1                          if N = 1
factorial( N ) =    ⎨
                    ⎩  N * factorial( N-1 )       otherwise
```

Recursive Method

- An *recursive method* is a method that contains a statement (or statements) that makes a call to itself.
- Implementing the factorial of N recursively will result in the following method.

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.

```
public int factorial( int N ) {

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


When Not to Use Recursion

- When recursive algorithms are designed carelessly, it can lead to very inefficient and unacceptable solutions.
- For example, consider the following:

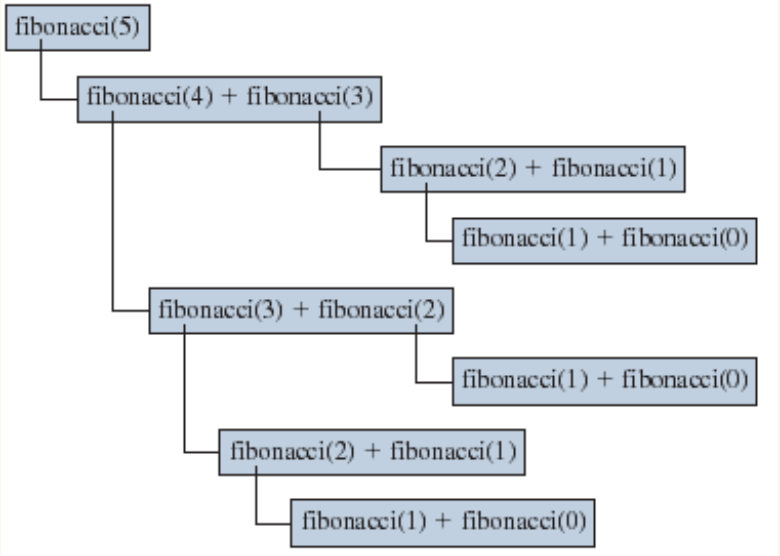
```
public int fibonacci( int N ) {

   if (N == 0 || N == 1) {
      return 1;

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

Excessive Repetition

- Recursive Fibonacci ends up repeating the same computation numerous times.



Nonrecursive Fibonacci

```
public int fibonacci( int N ) {

   int fibN, fibN1, fibN2, cnt;

   if (N == 0 || N == 1 ) {
      return 1;

   } else {

      fibN1 = fibN2 = 1;
      cnt = 2;
      while ( cnt <= N ) {
          fibN = fibN1 + fibN2; //get the next fib no.
          fibN1 = fibN2;
          fibN2 = fibN;
          cnt ++;
      }
      return fibN;
   }
}
```

When Not to Use Recursion

- In general, use recursion if
  - A recursive solution is natural and easy to understand.
  - A recursive solution does not result in excessive duplicate computation.
  - The equivalent iterative solution is too complex.