# UEE1302
# Introduction to Computers and Programming

C_Lecture 06:

Recursion

**C: How to Program 7th ed.**

# Agenda

- Recursive Calls
  - *n* factorial
  - Fibonacci Series
  - Hanoi Tower Game
- Recursive vs. Iterative

# Recursive Calls

- Functions can call themselves!
  - called recursive call (or recursion)
- Recursion
  - is very simple to express a complicated computation recursively.
  - can be converted to non-recursive functions
- Developing a recursive function
  - **base step**: when the function does not call itself again => stop condition
  - **recursive step**: compute the return value the help of the function itself => call itself

# Base Step in Recursion

- The base step corresponds to a case in which you've known the answer
  - the function returns the value immediately
  - or can easily compute the answer
  - typically, f(0), f(1) and etc.
- If you don't know a base step, you can't use recursion!
  - probably do NOT understand the problem
  - often cause infinite execution of the program if no base step => never stop!

# Recursive Step in Recursion

- Use the recursive call to solve a **sub-problem**
  - the parameters must be *different*
  - typically the input range becomes smaller => otherwise, the recursive call will get us no closer to the solution
- Need to do something besides just making recursive calls repeatedly

# *n* factorial

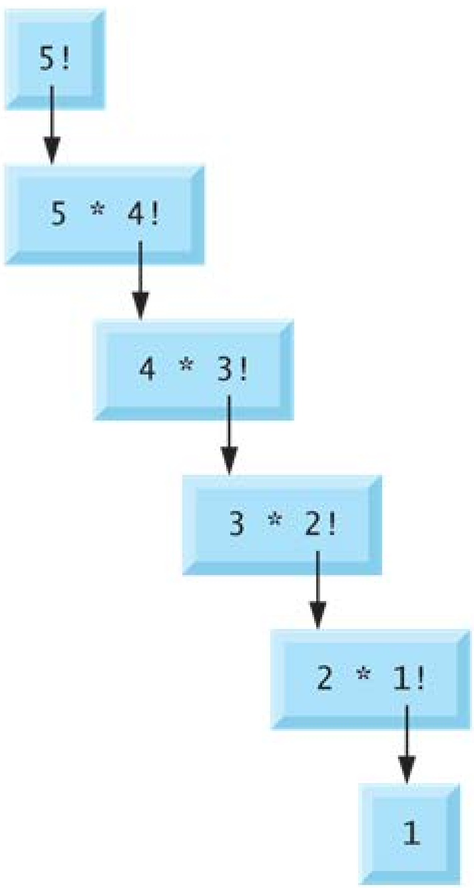
- The factorial of a nonnegative integer n, written n!, is the product $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$.
  - $1! = 1, \quad 0! = 1$
- The factorial of an integer, `number`, greater than or equal to 0 can be calculated iteratively (non-recursively) using a `for` statement as follows:

```
factorial = 1;
for ( counter = number; counter >= 1; counter--)
{
    factorial *= counter;
}
```

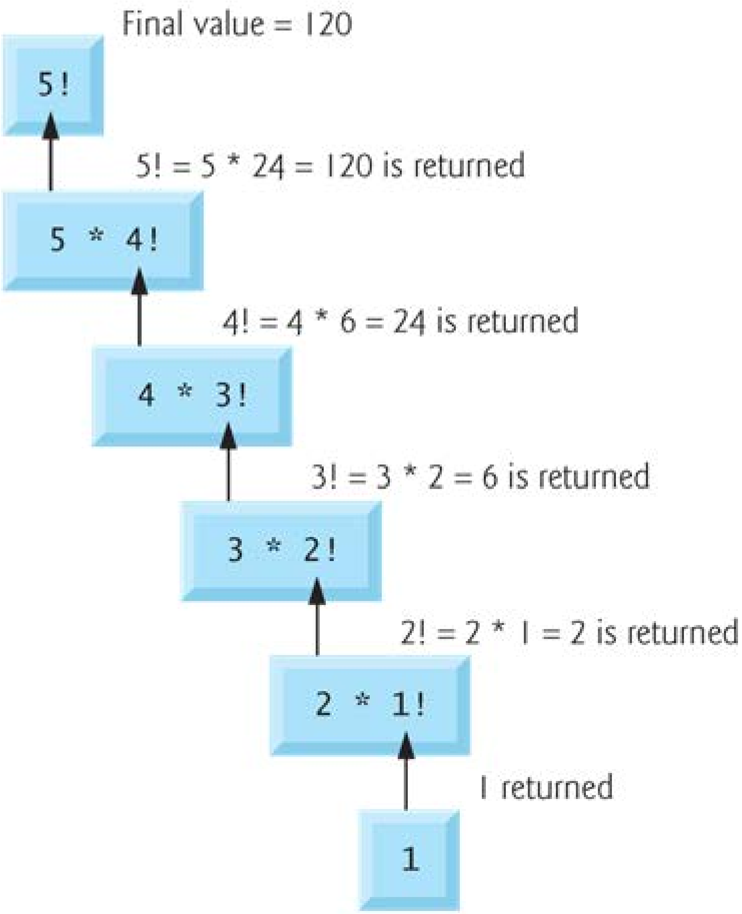
# *n* factorial (cont.)

- A recursive definition of the factorial function is arrived at by observing the following relationship:
    - $n! = n \cdot (n - 1)!$
- For example, 5! is clearly equal to 5 * 4! as is shown by the following:
    - $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
    - $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
    - $5! = 5 \cdot (4!)$

# Recursive Evaluation of 5!




(a) Sequence of recursive calls.




(b) Values returned from each recursive call.

# Recursive factorial Function

```c
// Fig. 5.18: fig05_18.c
// Recursive factorial function.
#include <stdio.h>

// function prototype
unsigned long long int factorial( unsigned int number );

int main ( void )
{
    unsigned int i;
    for ( i = 0; i <= 10; i++ )
        printf( "%u! = %llu\n", i, factorial(i) );
    return 0;
}
```

# Recursive factorial Function (cont.)

```
Unsigned long long int factorial ( unsigned int number )
{
    //base case
    if ( number <= 1 )
        return 1;
    else  //recursive step
        return  (number * factorial ( number - 1 ));
}
```

# Recursive factorial Function (cont.)

- screen output

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

# Fibonacci Series

- The Fibonacci series

  0, 1, 1, 2, 3, 5, 8, 13, 21, …

  - begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The Fibonacci series can be defined recursively as follows:

  fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

# Demonstrating the Fibonacci Series

```
// Fig. 5.19: fig05_19.c
// Fibonacci Series.
#include <stdio.h>
// function prototype
Unsigned long long int fibonacci ( unsigned int n );

int main ( void )
{
    unsigned long long int results;
    unsigned int number;
    printf( "Enter an integer: ");
    scanf("%u", &number);
    result = fibonacci( number );
    printf( "Fibonacci( %u ) = %llu\n", number, result);

    return 0;
}
```

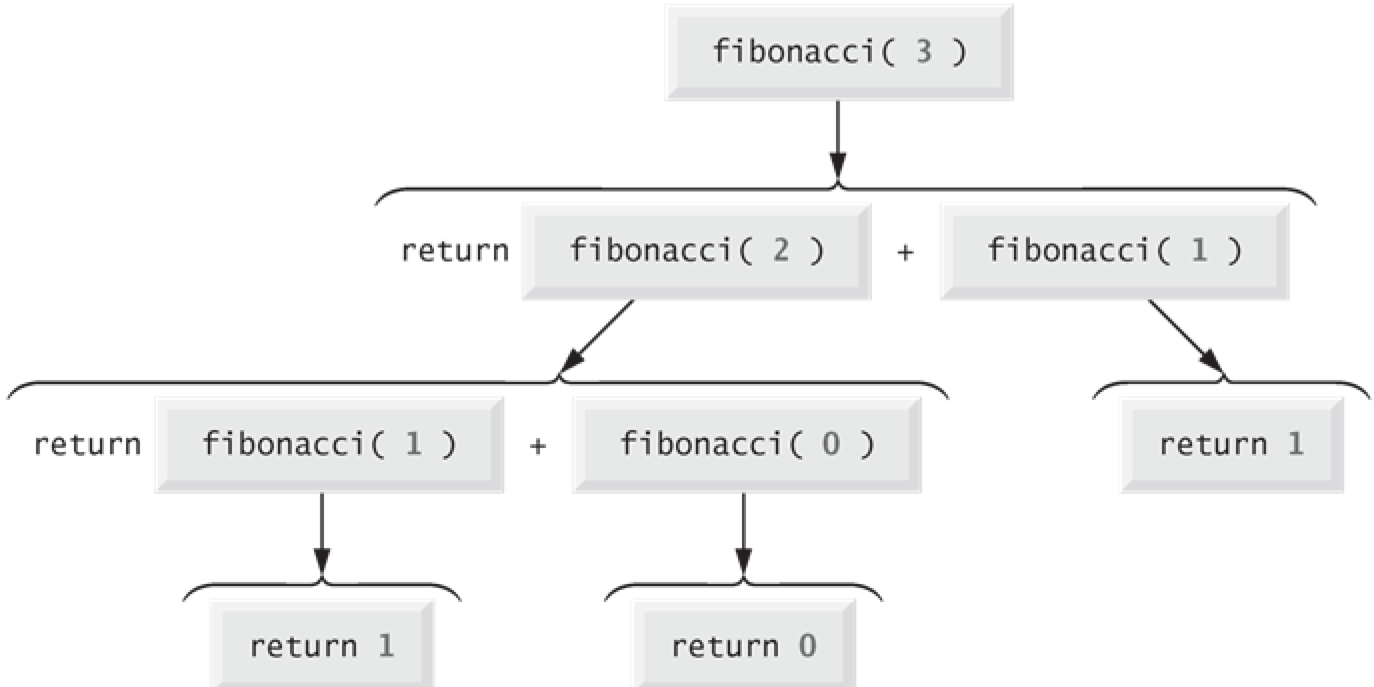
# Demonstrating the Fibonacci Series

```
Unsigned long long int fibonacci ( unsigned int n )
{
    //base case
    if ( ( n == 0) || (n == 1) )
        return n;
    else  //recursive step
        return fibonacci( n – 1) + fibonacci (n – 2);
}
```

- screen output

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

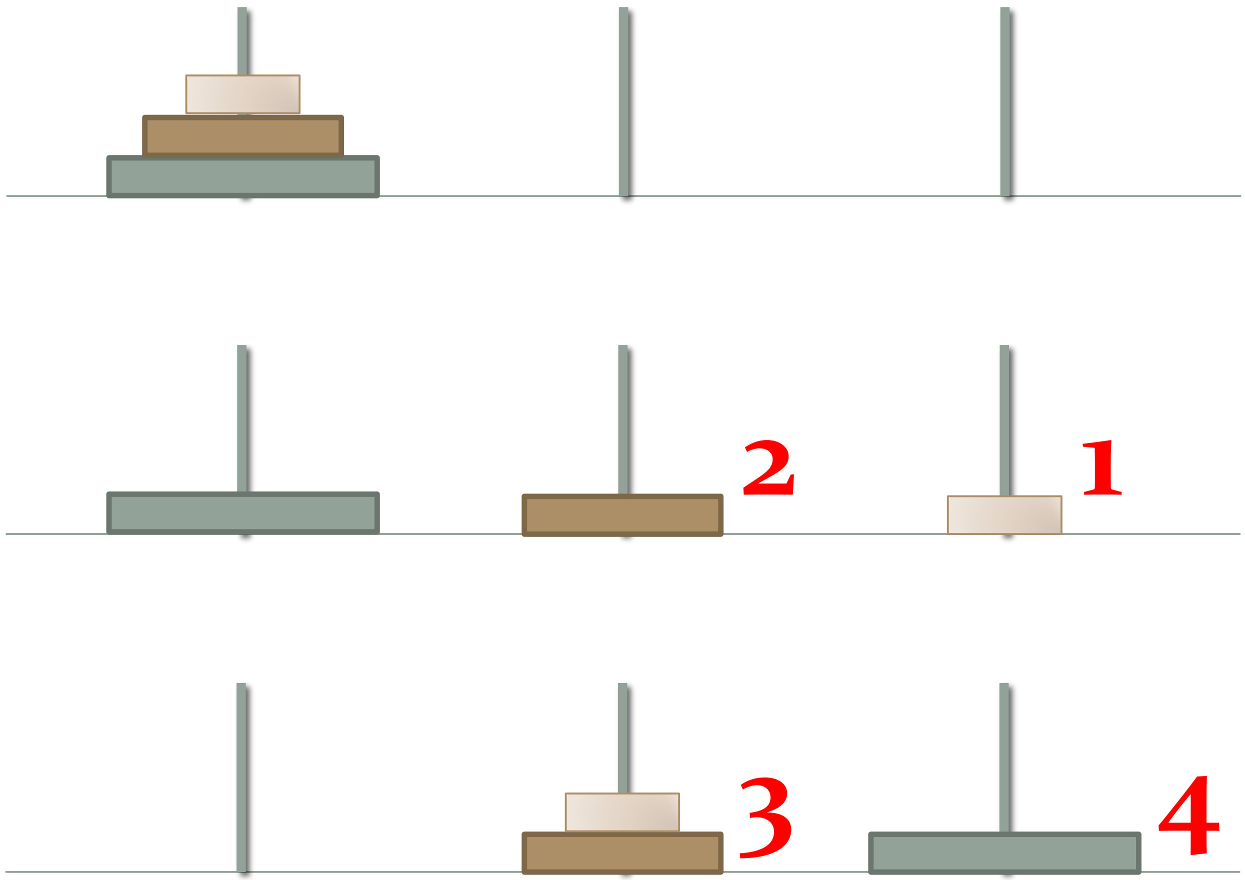
# Set of Recursive Calls to Function Fibonacci



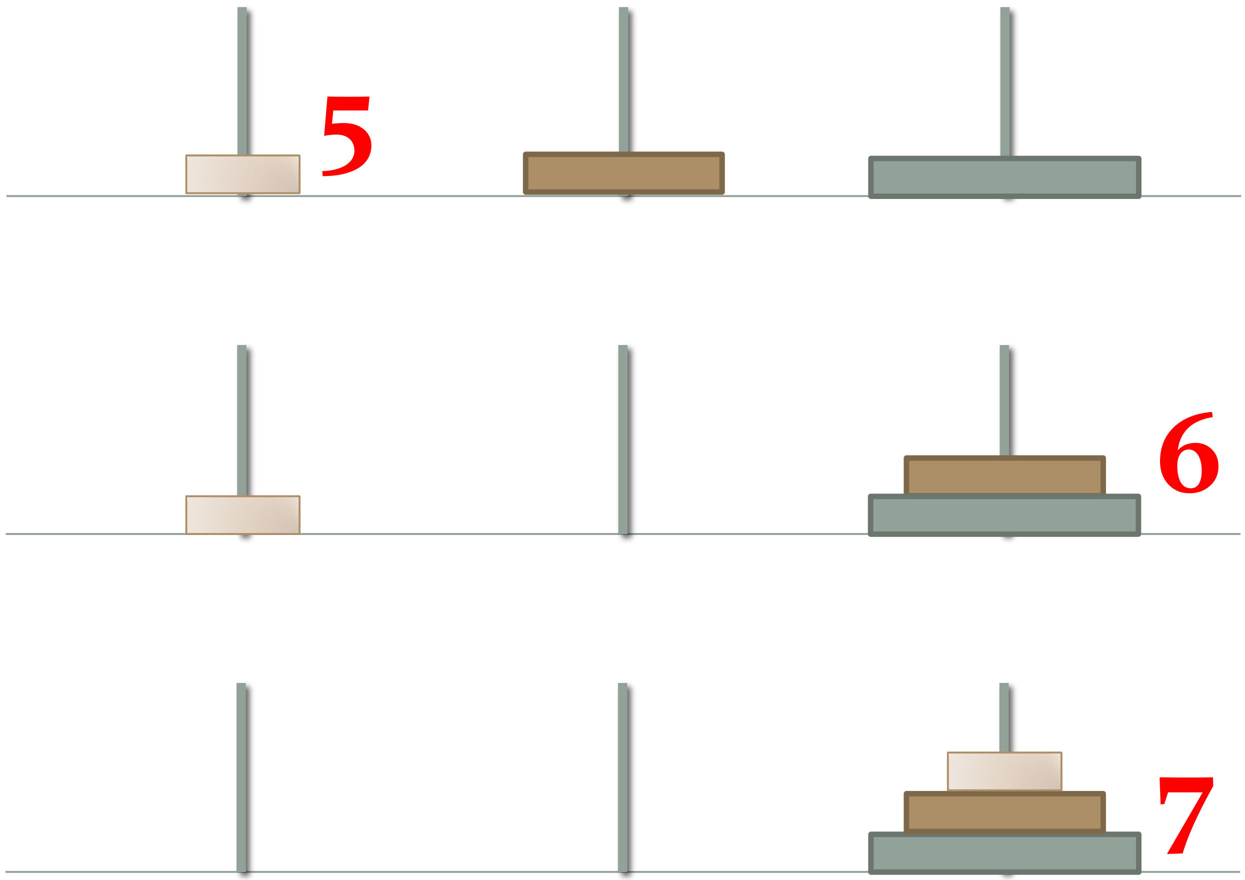
# Hanoi Tower Game

- A popular Math game in Europe, 19th century:
  - consists of three pegs (poles), and a number of disks of different sizes
  - objective is to move the entire stack of disks from 1st peg to 3rd peg
- Game rules:
  - Only one disk may be moved at a time.
  - Each move consists of taking the upper disk from one of the pegs and sliding it onto another peg
  - No disk can be placed on top of a smaller disk

# Example of Hanoi Tower

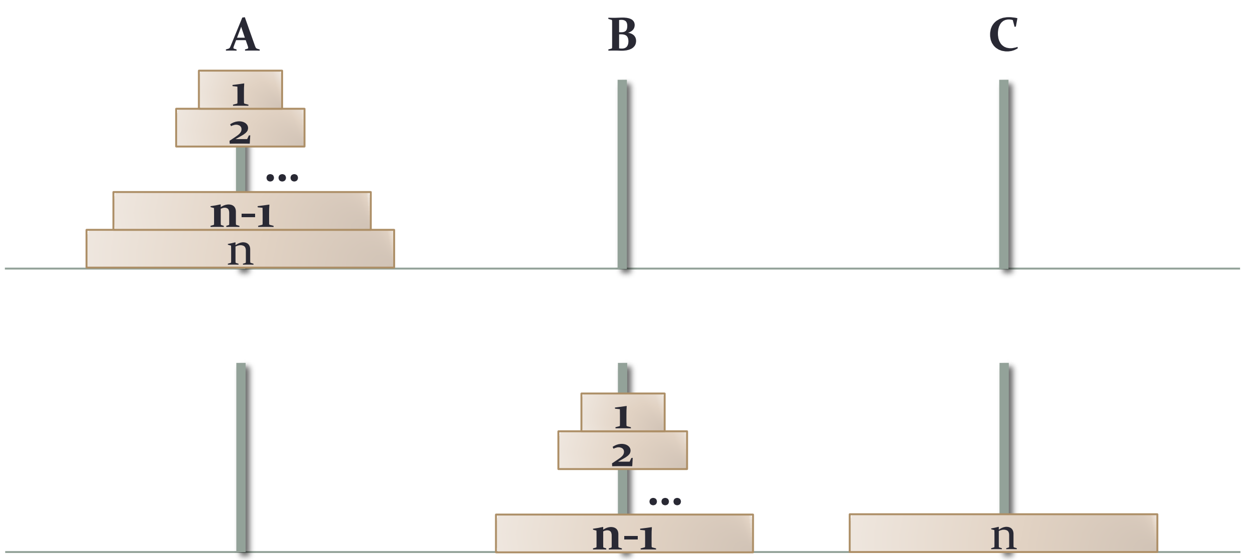


# Example of Hanoi Tower (cont.)



# Recursive Step for Hanoi Tower

- Key idea:
  - Move 1..(n-1) disks from peg A to peg B
  - Move n disk from peg A to Peg C
  - Move 1..(n-1) disks from peg B to peg C



# Pseudocode for Hanoi Tower

```
// This is only the pseudocode for Hanoi Tower
// C code can be developed easily
HanorTower(n, A, B, C)
{
    if ( n == 1)
    {
        move disk 1 from peg A to peg C;
    }
    else
    {
        HanoiTower(n-1, A, C, B);
        move disk n from peg A to peg C;
        HanoiATower(n-1, B, A, C);
    }
}
```

# Recursion vs. Iteration

- Both iteration and recursion are based on a control statement:
  - Iteration uses a repetition structure;
  - Recursion uses a selection structure
- Both iteration and recursion involve repetition:
  - Iteration explicitly uses a repetition structure;
  - Recursion achieves repetition through repeated function calls

# Recursion vs. Iteration (cont.)

- Iteration and recursion both involve a termination test:
  - Iteration terminates when the loop-continuation condition fails;
  - Recursion terminates when a base case is recognized
- Iteration with counter-controlled repetition and recursion both gradually approach termination:
  - Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail;
  - Recursion produces simpler versions of the original problem until the base case is reached

# Recursion vs. Iteration (cont.)

- Both iteration and recursion can occur infinitely:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false;
  - Infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case

# Factorial Function (Recursion)

```
// Fig. 5.14: fig05_14.c
// Recursive factorial function.
#include <stdio.h>

// function prototype
long factorial( long number );

int main ( void )
{
    int i;
    for ( i = 0; i <= 10; i++)
        printf( "%2d! = %d\n", i, factorial(i) );
    return 0;
}
```

# Factorial Function (Recursion) (cont.)

```
long factorial ( long number )
{
    //base case
    if ( number <= 1 )
        return 1;
    else  //recursive step
        return  (number * factorial ( number - 1 ));
}
```

# Factorial Function (Iterative)

```
#include <stdio.h>

// function prototype
long factorial( long number );

int main ( void )
{
    int i;
    for ( i = 0; i <= 10; i++)
        printf( "%2d! = %d\n", i, factorial(i) );
    return 0;
}
```

# Factorial Function (Iterative) (cont.)

```
long factorial ( long number )
{
    long result = 1;
    long i;
    for ( i = number; i >= 1; i--)
        result *= i;
    return result;
}
```

# Summary

- Recursion express a complicated computation recursively
  - a complete recursion consists of (1) base step and (2) recursive step
  - recursion for Fibonacci sequence
  - Hanoi Tower Game
- Recursive vs. Iterative