

Functions

- C supports the use of library functions, which are used to carry out a number of commonly used operations or calculations .
- C also allows programmers to define their own functions for carrying out various individual tasks.
- Use of programmer-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose. Thus a C program
- can be *modularized through the intelligent use of such functions.*

Advantages to this modular approach

1. *Use of a function avoids the need for redundant (repeated) programming of the same instructions*
 - For example, many programs require that a particular group of instructions be accessed repeatedly, from several different places within the program. The repeated instructions can be placed within a single function, which can then be accessed whenever it is needed.
 - Moreover, a different set of data can be transferred to the function each time it is accessed.

- 2. *logical clarity* resulting from the decomposition of a program into several concise functions, where each function represents some well-defined part of the overall problem.
- Such programs are easier to write and easier to debug.

- Use of functions also enables a programmer to build a *customized library* of frequently used routines or of routines containing system-dependent features.
- Each routine can be programmed as a separate function and stored within a special library file.

Overview

- how functions are defined and how they are accessed from various places within a C program.
- consider the manner in which information is passed to a function.
- Our discussion will include the use *of function prototypes*, as recommended by the current ANSI standard.
- And finally, we will discuss an interesting and important programming technique known as *recursion*.

What is a C Function?

- *A function* is a self-contained program segment that carries out some specific, well-defined task.
- A function is invoked(called) by its name and parameters.
 - No two functions have the same name and parameter types in your C program.
 - The communication between the function and invoker is through the parameters and the return value.

- A function is independent:
 - It is “completely” self-contained.
 - It can be called at any places of your code and can be ported to another program.
 - A function receives zero or more parameters, performs a specific task, and returns zero or one value.
- Functions make programs reusable and readable.

5.5 Function Definitions

- Function definition format

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- Function-name: any valid identifier
- Return-value-type: data type of the result (default **int**)
 - **void** – indicates that the function returns nothing
- Parameter-list: comma separated list, declares parameters
 - A type must be listed explicitly for each parameter unless, the parameter is of type **int**



5.5 Function Definitions

- Function definition format (continued)

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- Declarations and statements: function body (block)
 - Variables can be declared inside blocks (can be nested)
 - Functions can not be defined inside other functions
- Returning control
 - If nothing returned
 - **return;**
 - or, until reaches right brace
 - If something returned
 - **return** *expression* ;





Outline



1. Function prototype (3 parameters)

2. Input values

2.1 Call function

3. Function definition

```
1  /* Fig. 5.4: fig05 04.c
2     Finding the maximum of three integers */
3  #include <stdio.h>
4
5  int maximum( int, int, int );    /* function prototype */
6
7  int main()
8  {
9     int a, b, c;
10
11     printf( "Enter three integers: " );
12     scanf( "%d%d%d", &a, &b, &c );
13     printf( "Maximum is: %d\n", maximum( a, b, c ) );
14
15     return 0;
16 }
17
18 /* Function maximum definition */
19 int maximum( int x, int y, int z )
20 {
21     int max = x;
22
23     if ( y > max )
24         max = y;
25
26     if ( z > max )
27         max = z;
28
29     return max;
30 }
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

Program Output

2000 Prentice Hall
All rights reserved

5.6 Function Prototypes

- Function prototype
 - Function name
 - Parameters – what the function takes in
 - Return type – data type function returns (default **int**)
 - Used to validate functions
 - **Prototype only needed if function definition comes after use in program**
 - The function with the prototype

```
int maximum( int, int, int );
```

 - Takes in 3 **ints**
 - Returns an **int**
- Promotion rules and conversions
 - Converting to lower types can lead to errors



Example

- //Function prototype

- int max(int a, int b);

```
int main(){
```

```
    int x;
```

```
//Function calling
```

```
    x = max(5,8) ;
```

```
        x = max(x,7);
```

```
}
```

```
//Function definition
```

```
int max(int a, int b)
```

```
{
```

```
    return a>b?a:b;
```

```
}
```

- The arguments are called *formal arguments*, because they represent the names of data items that are transferred into the function from the calling portion of the program.
- The corresponding arguments in the function *reference* are called *actual arguments*, since they define the data items that are actually transferred.

- The remainder of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the ***body*** of the function.
- It should include one or more **return** statements, in order to return a value to the calling portion of the program.

- Information is returned from the function to the calling portion of the program via the return statement.

```
char lower_to_upper(char c1)          /* programmer-defined (
{
    char c2;

    c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
    return(c2);
}
```


Multiple returns

- A function definition can include multiple return statements, each containing a different expression.
- Functions that include multiple branches often require multiple returns.

```
char lower_to_upper(char c1)
{
    if (c1 >= 'a' && c1 <= 'z')
        return('A' + c1 - 'a');
    else
        return(c1);
}
```

Empty return

```
maximum(int x, int y)          /* determine 1
{
    int z;

    z = (x >= y) ? x : y;
    printf("\n\nMaximum value = %d", z);
    return;
}
```

The keyword **void** can be used as a type specifier when defining a function that does not return anything, or when the function definition does not include any arguments. The presence of this keyword is not mandatory, but it is good programming practice to make use of this feature.

void

```
void maximum(x, y)          /* determine the larger of two integer  
int x, y;  
{  
    int z;  
  
    z = (x >= y) ? x : y;  
    printf("\n\nMaximum value = %d", z);  
    return;  
}
```

Write a function to find factorial of a given no

- The factorial of a positive integer quantity, n , is defined as $n! = 1 \times 2 \times 3 \times \dots \times n$.
- Thus, $2! = 1 \times 2 = 2$;
- $3! = 1 \times 2 \times 3 = 6$;
- $4! = 1 \times 2 \times 3 \times 4 = 24$; and so on.

Factorial of a number

```
long int factorial(int n)
{
    int i;
    long int prod = 1;

    if (n > 1)
        for (i = 2; i <= n; ++i)
            prod *= i;
    return(prod);
}
```

ACCESSING A FUNCTION

- A function can be *accessed* (i.e., *called*) by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas.
- If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function

Some Examples

- Function Prototype Examples

```
double squared (double number);
```

```
void print_report (int);
```

```
int get_menu_choice (void);
```

void parameter list means
it takes no parameters

- Function Definition Examples

```
double squared (double number)
```

```
{
```

```
    return (number * number);
```

```
}
```

```
void print_report (int report_number)
```

```
{
```

```
    if (report_number == 1)
```

```
        printf("Printer Report 1");
```

```
    else
```

```
        printf("Not printing Report 1");
```

```
}
```

return type void means
it returns nothing

- The arguments appearing in the function call are referred to as *actual arguments*, in contrast to the formal arguments that appear in the first line of the function definition.
- The actual arguments may be expressed as constants, single variables, or more complex
- Expressions and must be of the same data type as its corresponding formal

- If the function returns a value, the function access is often written as an assignment statement; e.g., **y = polynomial(x);**
- This function access causes the value returned by the function to be assigned to the variable **y**.
- On the other hand, if the function does not return anything, the function access appears by itself; e.g., **display(a, b, c);**

Largest of Three Integer Quantities

```
#include <stdio.h>

int maximum(int x, int y)
{
    int z;
    z = (x >= y) ? x : y;
    return(z);
}

main()
{
    int a, b, c, d;

    /* read the integer quantities */
    printf("\na = ");
    scanf("%d", &a);
    printf("\nb = ");
    scanf("%d", &b);
    printf("\nc = ");
    scanf("%d", &c);

    /* calculate and display the maximum value */
    d = maximum(a, b);
    printf("\n\nmaximum = %d", maximum(c, d));
}
```

Function Prototype

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

```
main()
```

```
{
```

```
    int a, b, c, d;
```

```
    /* read the integer quantities */
```

```
    printf("\na = ");
```

```
    scanf("%d", &a);
```

```
    printf("\nb = ");
```

```
    scanf("%d", &b);
```

```
    printf("\nc = ");
```

```
    scanf("%d", &c);
```

```
    /* calculate and display the maximum value */
```

```
    d = maximum(a, b);
```

```
    printf("\n\nmaximum = %d", maximum(c, d));
```

```
}
```

```
int maximum(int x, int y)
```

```
{
```

```
    int z;
```

```
    z = (x >= y) ? x : y;
```

```
    return(z);
```

```
}
```

FUNCTION PROTOTYPES

- Let the main appears ahead of the programmer-defined function definition. In such situations the function access (within **main**) will precede the function definition.
- This can be confusing to the compiler.
- The compiler is first alerted to the fact that the function being accessed will be defined later in the program.
- *A function prototype* is used for this

FUNCTION PROTOTYPES

contd..

- are usually written at the beginning of a program, ahead of any programmer-defined functions (including `main`). The general form of a function prototype is

data-type name(type 1 arg 1, type 2 arg 2, . . . , type n arg n);

PASSING ARGUMENTS TO A FUNCTION

- When a single value is passed to a function via an actual argument, the value of the actual argument is *copied* into the function. Therefore, *the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change.*
- This procedure for passing the value of an argument to a function is known as *passing by value.*

```
#include <stdio.h>

void modify(int a);      /* function prototype */

main()
{
    int a = 2;

    printf("\na = %d (from main, before calling the function)", a);
    modify(a);
    printf("\n\na = %d (from main, after calling the function)", a);
}

void modify(int a)
{
    a *= 3;
    printf("\n\na = %d (from the function, after being modified)", a);
    return;
}
```

RECURSION

- *Recursion* is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.
- Many iterative (i.e., repetitive) problems can be written in this form

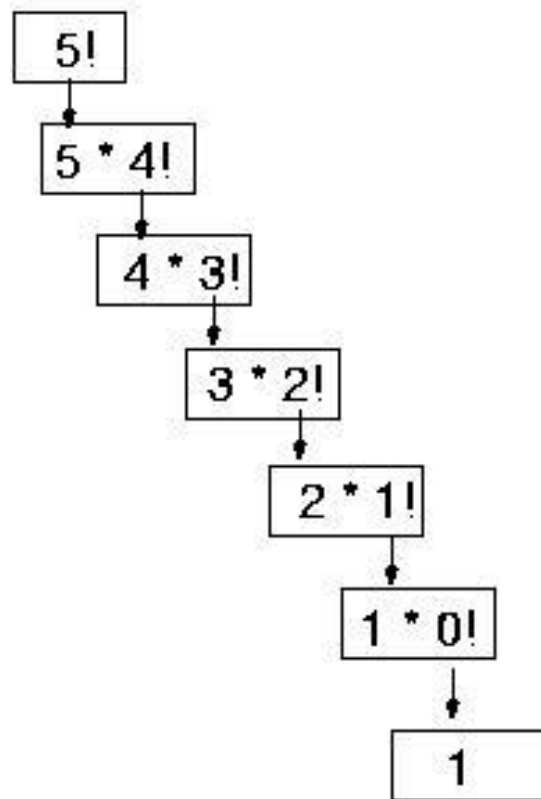
- To solve a problem recursively, two conditions must be satisfied.
- First, the problem must be written in a recursive form.
- Second, the problem statement must include a stopping condition

- For example, we wish to calculate the factorial of a positive integer quantity.

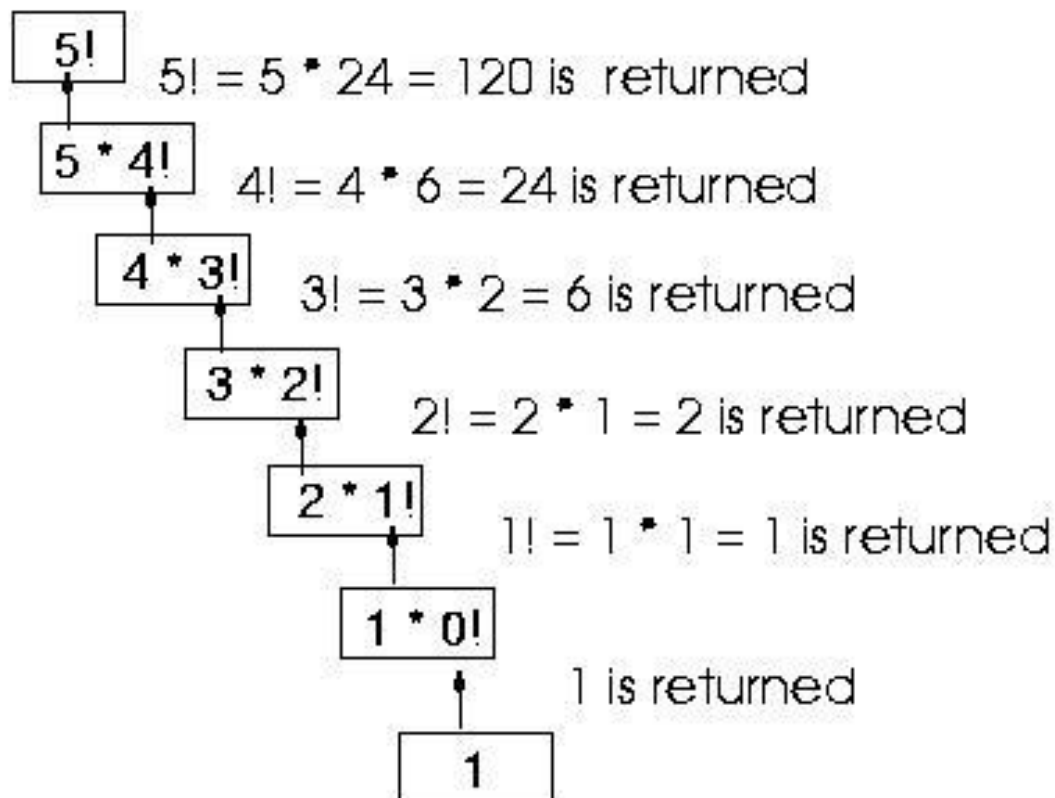
$$n! = 1 \times 2 \times 3 \times \dots \times n$$

OR

- $n! = n \times (n - 1)!$ - This is recursive statement of the problem.
- Also, we know that $1! = 1$ by definition. It provides a stopping condition for the recursion.



Final value = 120



Coding the factorial function

- Recursive implementation

```
int Factorial(int n)
{
    if (n==0) // base case
        return 1;
    else
        return n * Factorial(n-1);
}
```

Recursion vs. iteration

- Iteration can be used in place of recursion
 - An iterative algorithm uses a *looping construct*
 - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in *shorter*, more easily understood source code

```

#include <stdio.h>

long int factorial(int n);          /* function prototype */

main()
{
    int n;

    /* read in the integer quantity */

    printf("n = ");
    scanf("%d", &n);

    /* calculate and display the factorial */

    printf("n! = %ld\n",  factorial(n));
}

long int factorial(int n)          /* calculate the factorial */
{
    if (n <= 1)
        return(1);
    else
        return(n * factorial(n - 1));
}

```

How it works

- When a recursive program is executed, the recursive function calls are not executed immediately.
- Rather, they are placed on a *stack* until the condition that terminates the recursion is encountered.
- The function calls are then executed in reverse order, as they are “popped” off the stack.

Function calls will proceed in the following order

$$n! = n \times (n - 1)!$$

$$(n - 1)! = (n - 1) \times (n - 2)!$$

$$(n - 2)! = (n - 2) \times (n - 3)!$$

.....

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

The actual values will then be returned in the following reverse order.

$$1! = 1$$

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

$$3! = 3 \times 2! = 3 \times 2 = 6$$

$$4! = 4 \times 3! = 4 \times 6 = 24$$

.....

$$n! = n \times (n - 1)! = \dots$$

This reversal in the order of execution is a characteristic of all functions that are executed recursively

Printing Backwards

- reads in a line of text on a character-by-character basis, and then displays the characters in reverse order

```

/* read a line of text and write it out backwards, using recursion */
#include <stdio.h>
#define EOLN  '\n'
void reverse(void);          /* function prototype */

main()
{
    printf("Please enter a line of text below\n");
    reverse();
}

void reverse(void)
/* read a line of characters and write it out backwards */
{
    char c;

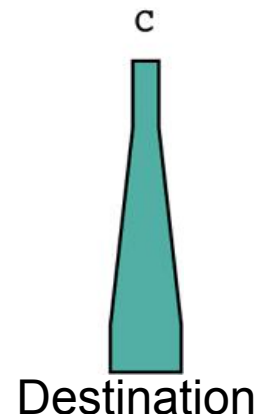
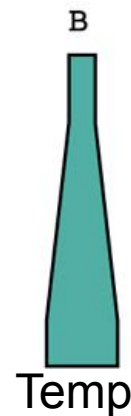
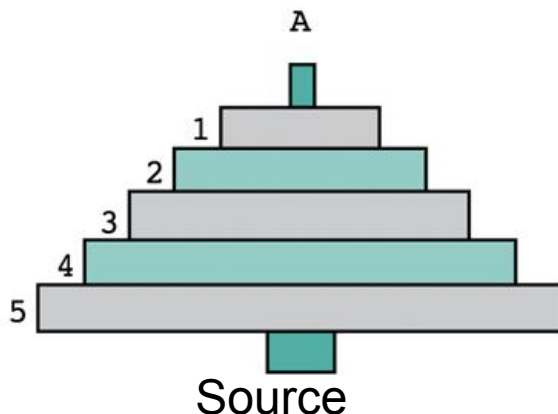
    if ((c = getchar()) != EOLN) reverse();
    putchar(c);
    return;
}

```

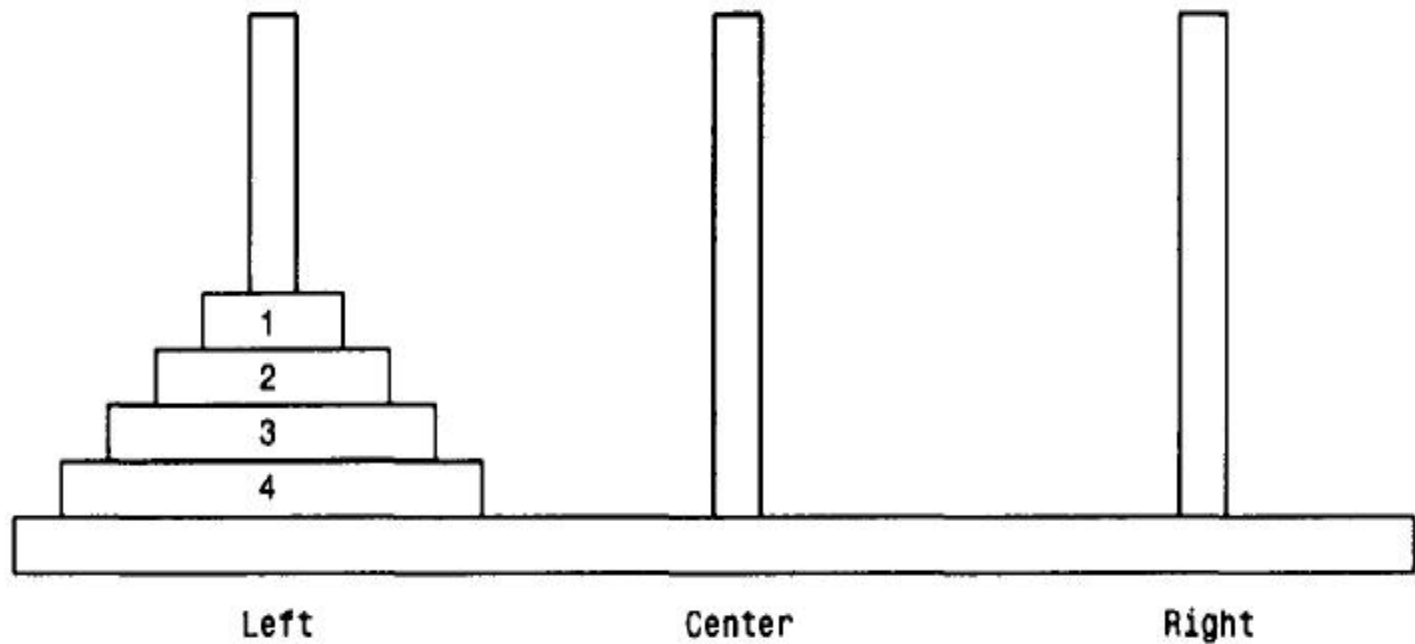
Each function call causes a new character (a new value for **c**) to be pushed onto the stack. Once the end of line is encountered, the successive characters are popped off the stack and displayed on a last-in, first-out basis. Thus, the characters are displayed in reverse order

A Classical Case: Towers of Hanoi

- The towers of Hanoi problem involves moving a number of disks (in different sizes) from one tower (or called “peg”) to another.
 - The constraint is that the larger disk can never be placed on top of a smaller disk.
 - Only one disk can be moved at each time
 - Assume there are three towers available.



The Towers of Hanoi



- Three poles and a number of different-sized disks.
- Each disk has a hole in the center, allowing it to be stacked around any of the poles.
- Initially, the disks are stacked on the leftmost pole in the order of decreasing size, i.e., the largest on the bottom and the smallest on the top

- The object of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on top of a smaller disk.
- Only one disk may be moved at a time, and each disk must always be placed around one of the poles.

Recursive Solution

- 1. Move the top $n - 1$ disks from the left pole to the center pole.
- 2. Move the n th disk (the largest disk) to the right pole.
- 3. Move the $n - 1$ disks on the center pole to the right pole.
- The problem can be solved in this manner for any value of n greater than 0 ($n=0$ represents a stopping condition).

```
/* the TOWERS OF HANOI - solved using recursion */  
#include <stdio.h>  
  
void transfer(int n, char from, char to, char temp);  
  
main()  
{  
    int n;  
  
    printf("Welcome to the TOWERS OF HANOI\n\n");  
    printf("How many disks? ");  
    scanf("%d", &n);  
    printf("\n");  
    transfer(n, 'L', 'R', 'C');  
}
```



```

void transfer(int n, char from, char to, char temp)

/* transfer n disks from one pole to another */

/* n    = number of disks
   from = origin
   to   = destination
   temp = temporary storage */

{
    if (n > 0)    {
        /* move n-1 disks from origin to temporary */
        transfer(n-1, from, temp, to);

        /* move nth disk from origin to destination */
        printf("Move disk %d from %c to %c\n", n, from, to);

        /* move n-1 disks from temporary to destination */
        transfer(n-1, temp, to, from);
    }
    return;
}

```

OUTPUT:

Welcome to the TOWERS OF HANOI

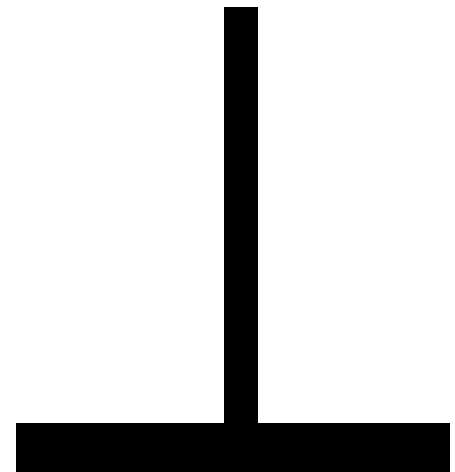
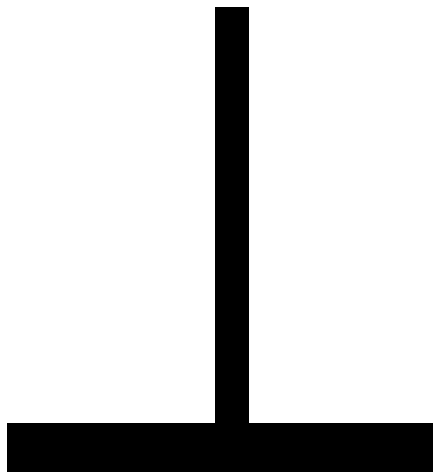
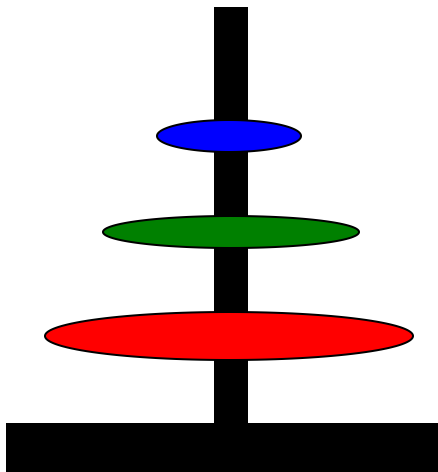
How many disks? 3

```

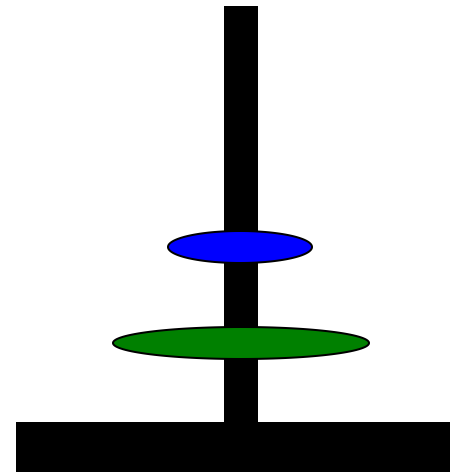
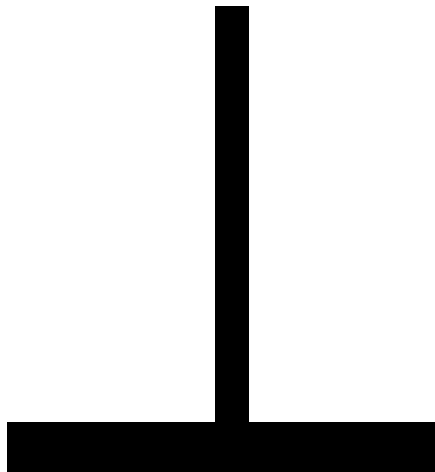
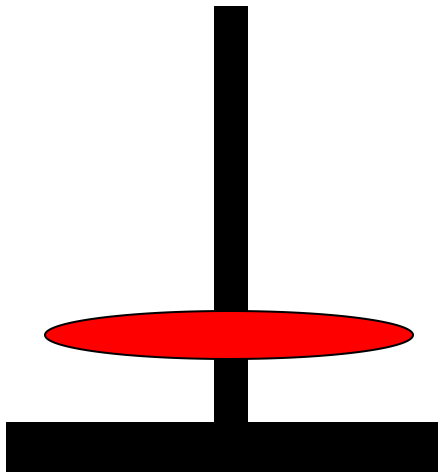
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R

```

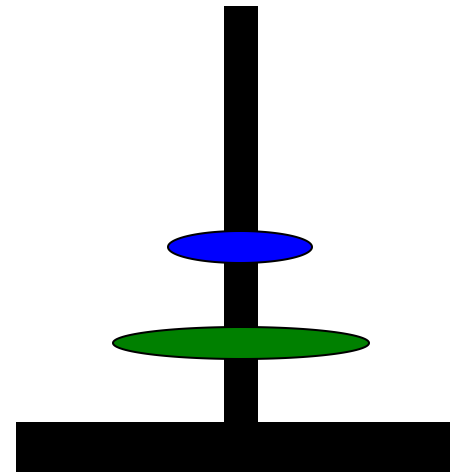
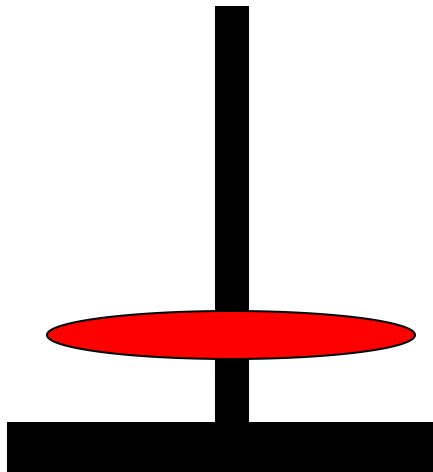
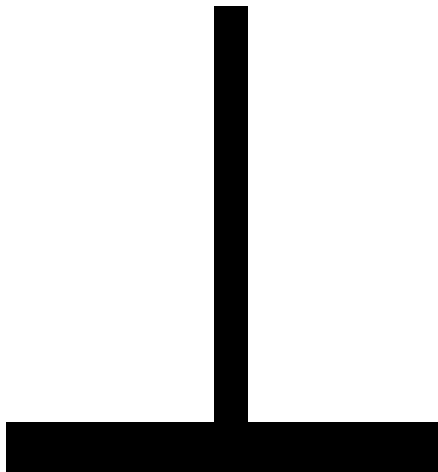
Recursive Solution



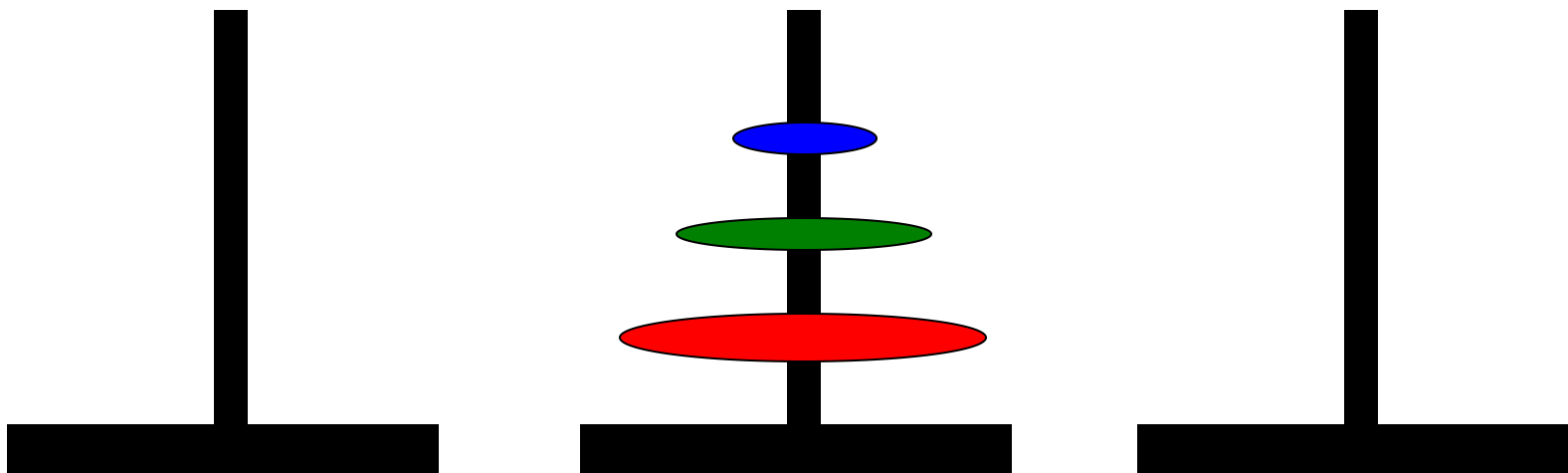
Recursive Solution



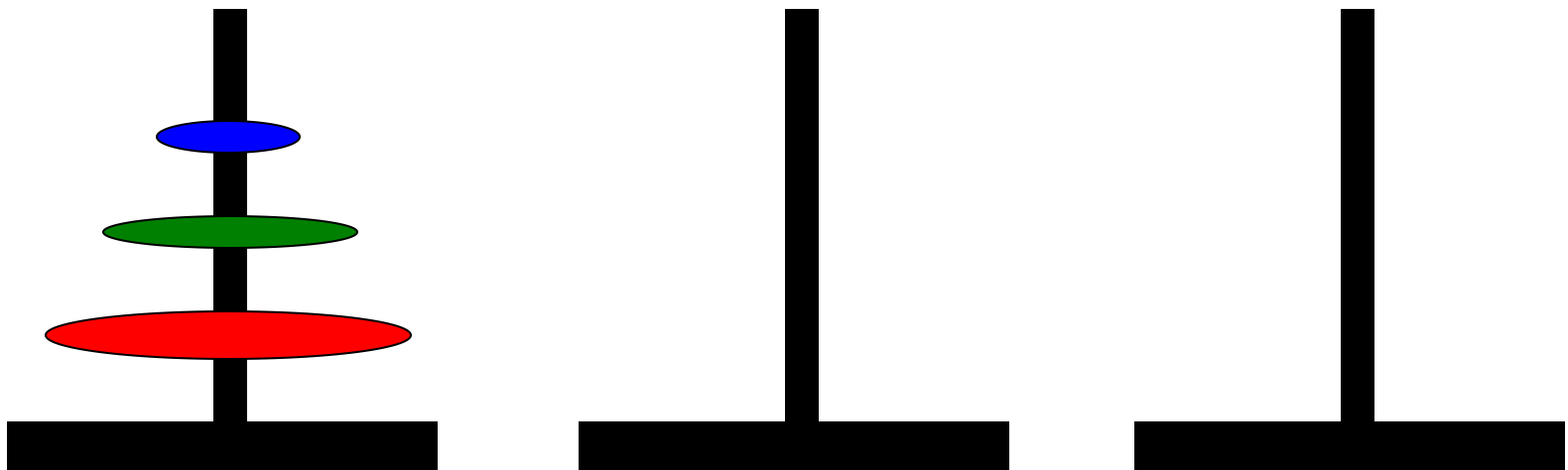
Recursive Solution



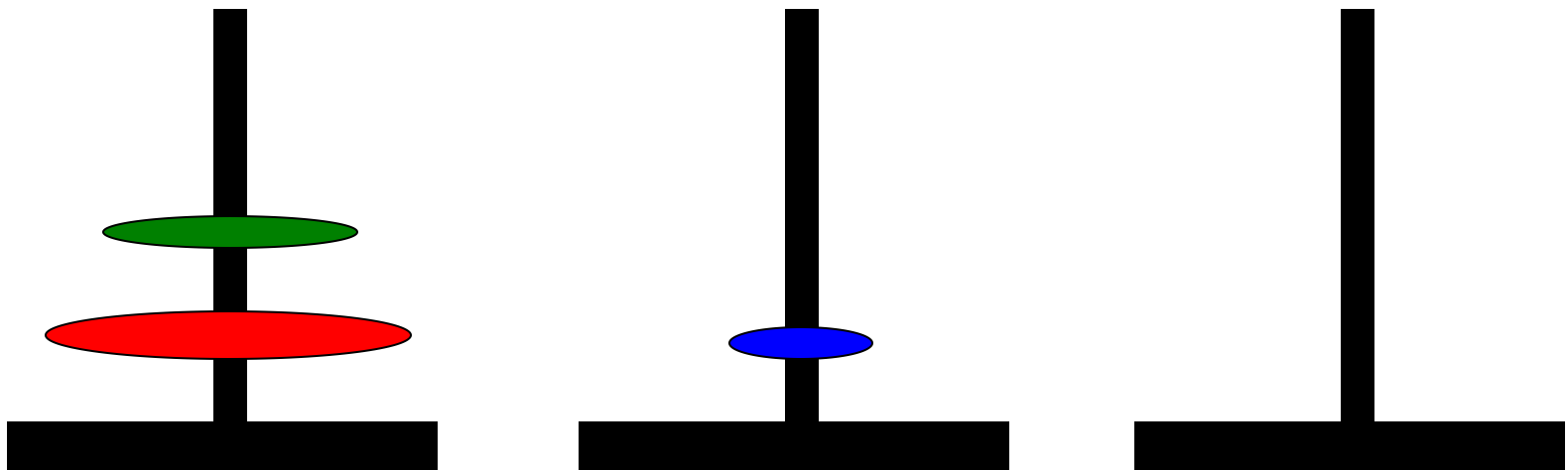
Recursive Solution



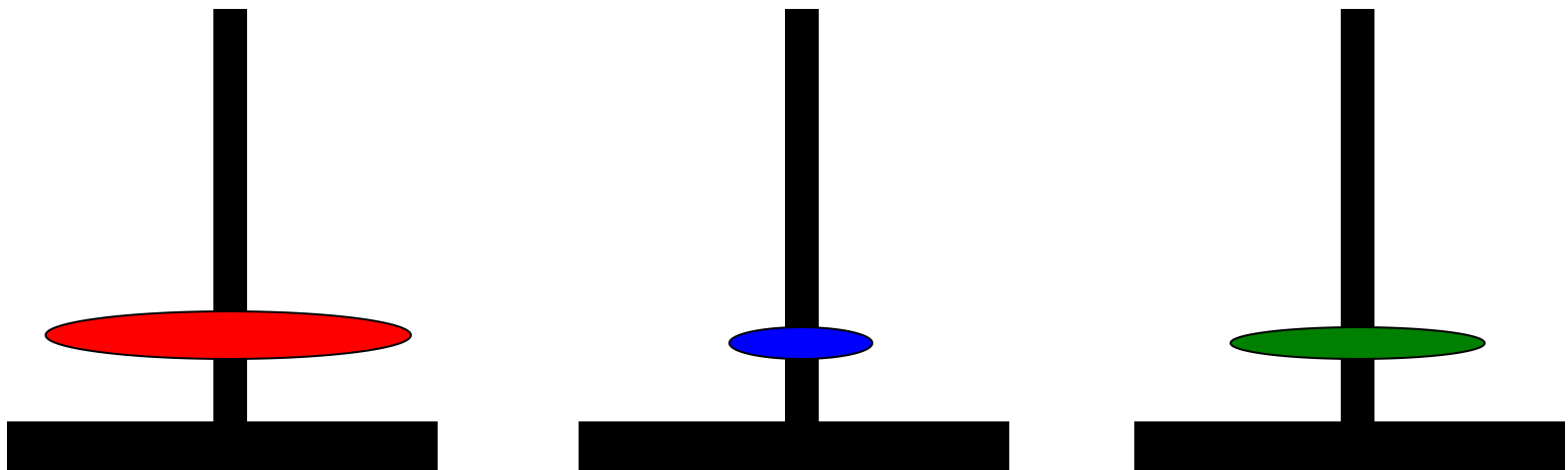
Tower of Hanoi



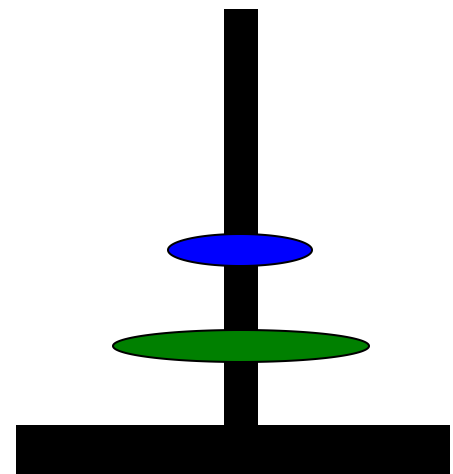
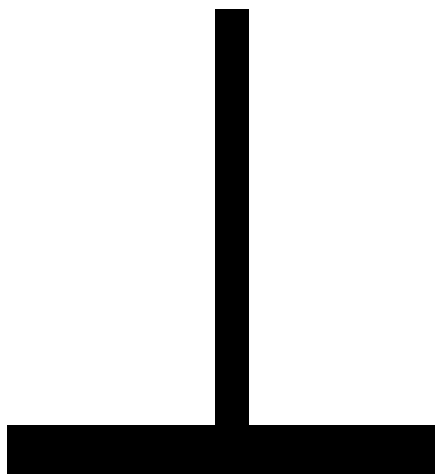
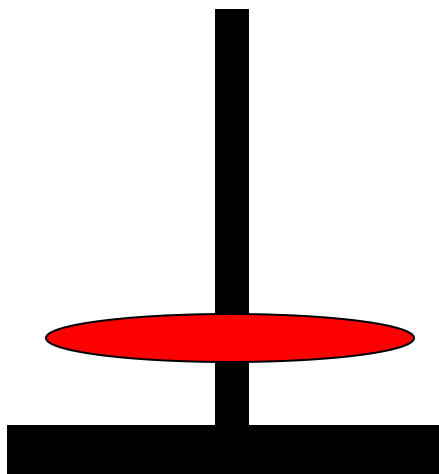
Tower of Hanoi



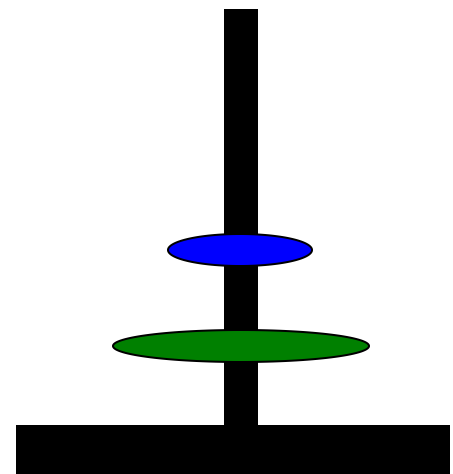
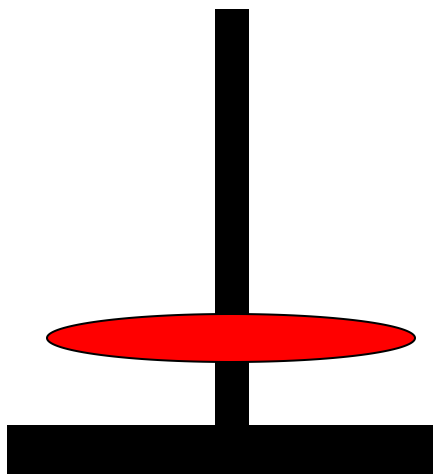
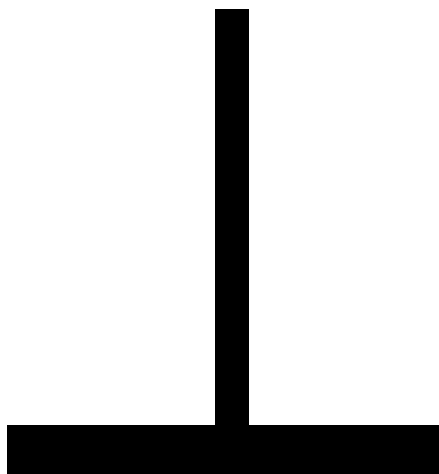
Tower of Hanoi



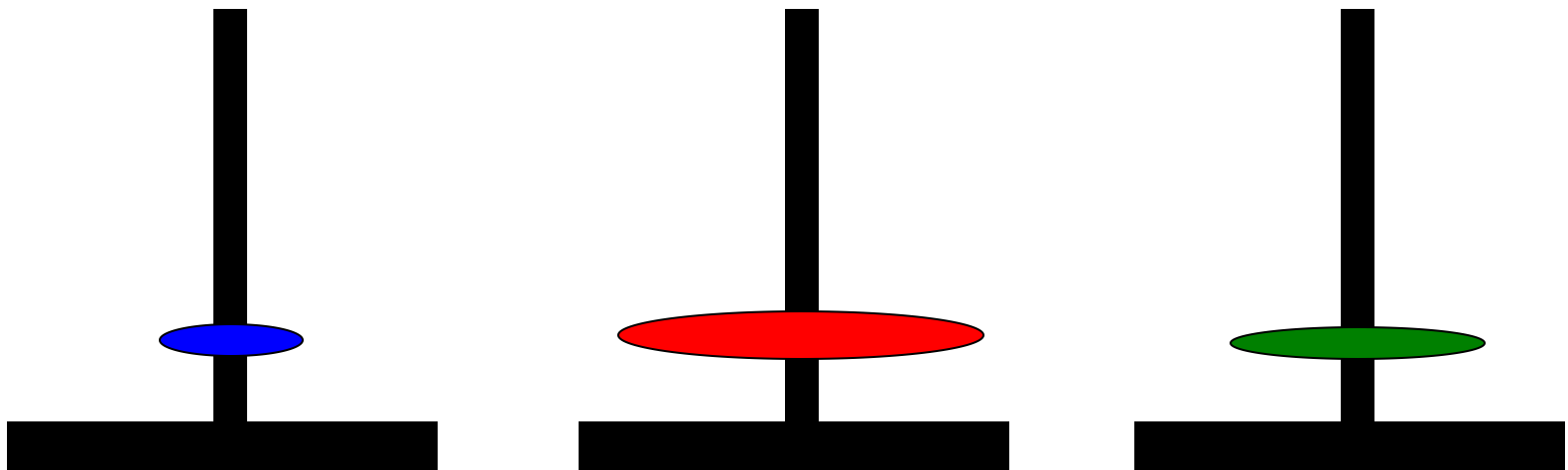
Tower of Hanoi



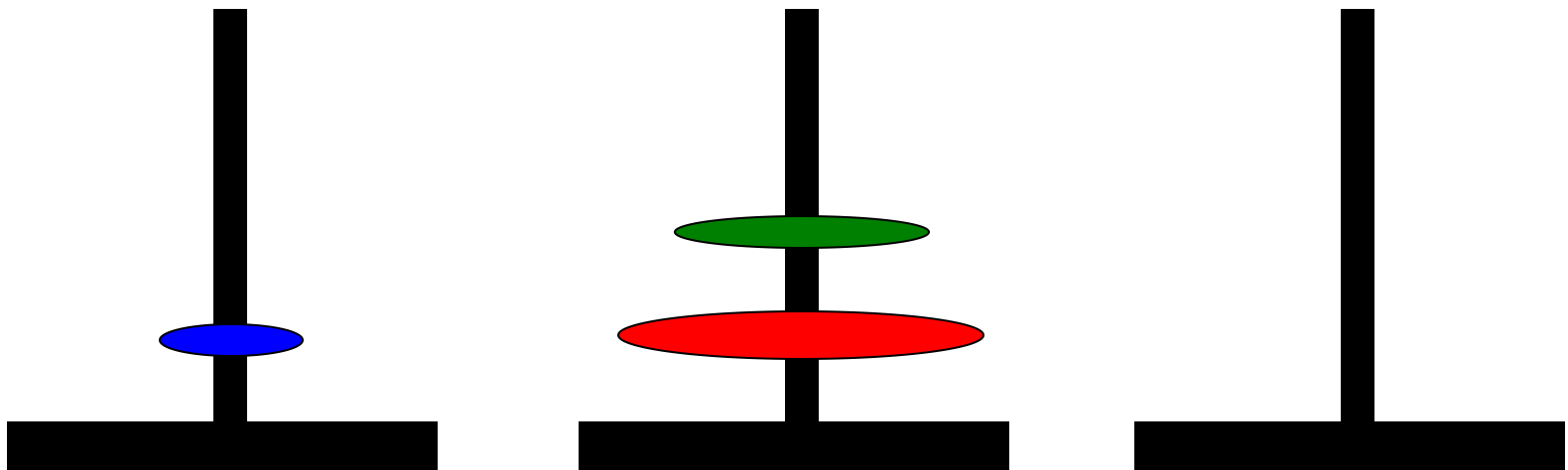
Tower of Hanoi



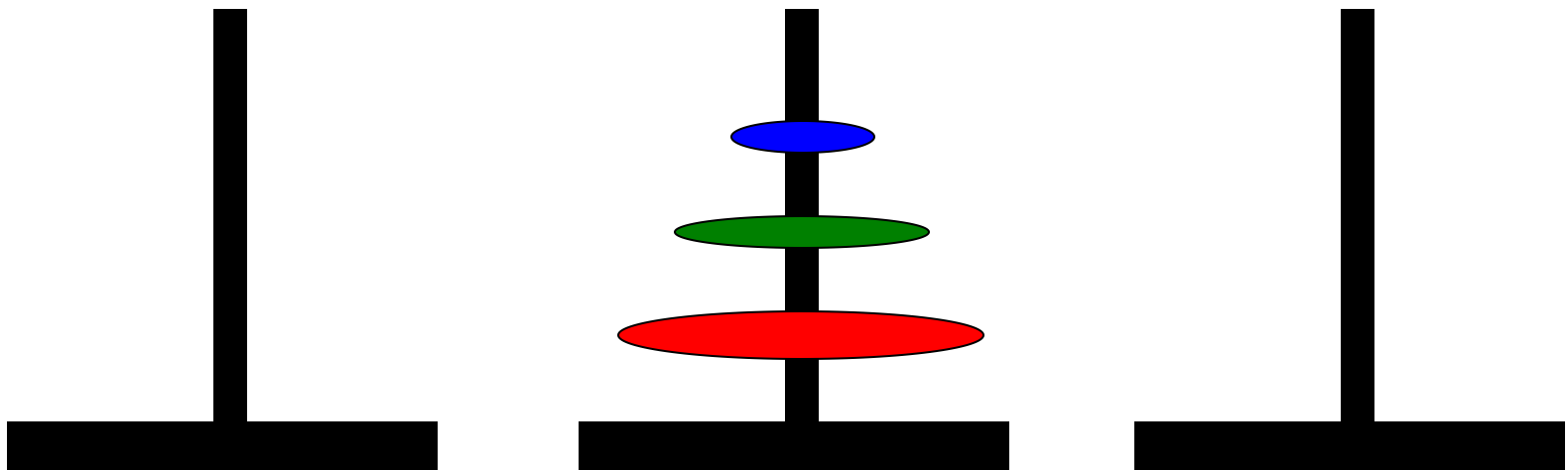
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Direct Computation Method

- Fibonacci numbers:

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

where each number is the sum of the preceding two.

Recursive definition:

$$- F(0) = 1;$$

$$- F(1) = 1;$$

$$- F(\text{number}) = F(\text{number}-1) + F(\text{number}-2);$$

Recursive fibonacci Function


- In our midterm, you are required to write a fibonacci function.
 - It can be easily solved by the recursive function

```
6. fibonacci(int n)
7. {
8.     int ans;
9.
10.    if (n == 1 || n == 2)
11.        ans = 1;
12.    else
13.        ans = fibonacci(n - 2) + fibonacci(n - 1);
14.
15.    return (ans);
16. }
```

Recursive Function to Count a Character in a String

- We can count the number of occurrences of a given character in a string.
 - e.g., the number of 's' in “Mississippi” is 4.

```
4. int
5. count(char ch, const char *str)
6. {
7.
8.     int ans;
9.
10.    if (str[0] == '\0')                /* simple case */
11.        ans = 0;
12.    else                                /* redefine problem using recursion */
13.        if (ch == str[0])              /* first character must be counted */
14.            ans = 1 + count(ch, &str[1]);
15.        else                            /* first character is not counted */
16.            ans = count(ch, &str[1]);
17.
18.    return (ans);
19. }
```



An Example of Recursive Function

- We can implement the multiplication by addition.

```
1.  /*
2.   * Performs integer multiplication using + operator.
3.   * Pre:  m and n are defined and n > 0
4.   * Post: returns m * n
5.   */
6.  int
7.  multiply(int m, int n)
8.  {
9.      int ans;
10.
11.      if (n == 1)
12.          ans = m;      /* simple case */
13.      else
14.          ans = m + multiply(m, n - 1); /* recursive step */
15.
16.      return (ans);
17. }
```

The simple case is " $m \cdot 1 = m$."

The recursive step uses the following equation:
" $m \cdot n = m + m \cdot (n - 1)$."

Recursive gcd Function

- e.g., the greatest common divisor (GCD) of two integers m and n can be defined recursively.
 - $\text{gcd}(m,n)$ is n if n divides m evenly;
 - $\text{gcd}(m,n)$ is $\text{gcd}(n, \text{remainder of } m \text{ divided by } n)$ otherwise.

```
4. #include <stdio.h>
5.
6.
7. /*
8.  * Finds the greatest common divisor of m and n
9.  * Pre: m and n are both > 0
10. */
11. int
12. gcd(int m, int n)
13. {
14.     int ans;
15.
16.     if (m % n == 0)
17.         ans = n;
18.     else
19.         ans = gcd(n, m % n);
20.
21.     return (ans);
22. }
```

(continued)