Recursive Functions

When a function calls itself

Pallab Dasgupta
Professor,
Dept. of Computer Sc & Engg

Recursion

A process by which a function calls itself repeatedly.

- Either directly.
    - F calls F.
- Or cyclically in a chain.
    - F calls G, G calls H, and H calls F.

Used for repetitive computations in which each action is stated in terms of a previous result.

fact(n) = n * fact (n-1)

Basis and Recursion

For a problem to be written in recursive form, two conditions are to be satisfied:

- **It should be possible to express the problem in recursive form.**
- **The problem statement must include a stopping condition**

```
fact(n)  =  1,                    if  n = 0     /* Stopping criteria */
         =  n * fact(n – 1),      if  n > 0     /* Recursive form */
```

Examples:

- **Factorial:**

  fact(0) = 1

  fact(n) = n * fact(n – 1), if n > 0

- **GCD:**

  gcd (m, m) = m

  gcd (m, n) = gcd (m%n, n), if m > n

  gcd (m, n) = gcd (n, n%m), if m < n

- **Fibonacci series (1,1,2,3,5,8,13,21,….)**

  fib (0) = 1

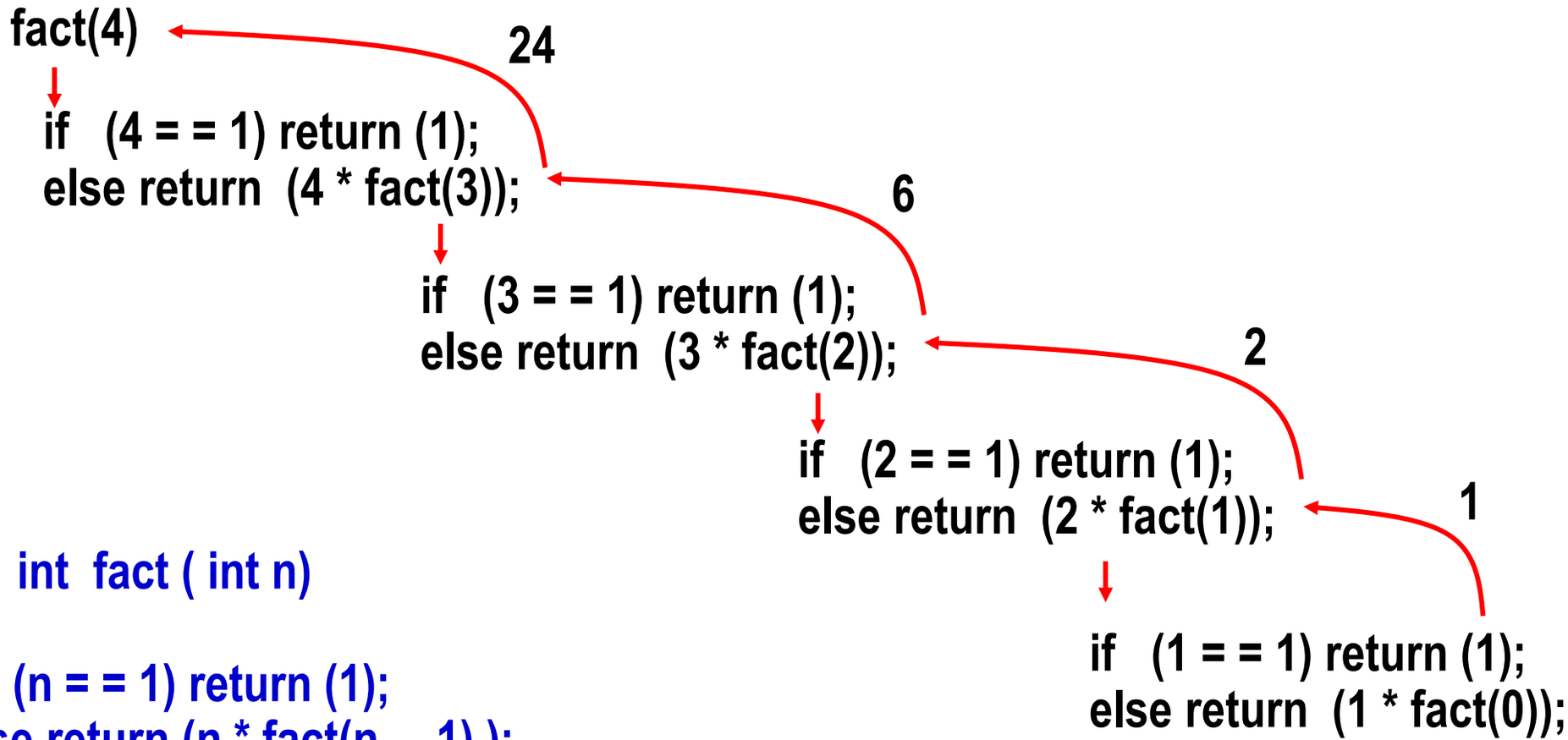
  fib (1) = 1

  fib (n) = fib (n – 1) + fib (n – 2), if n > 1

Example 1 :: Factorial

```
long  int  fact ( int n)
{
   if   (n = = 1)
      return (1);
   else
      return  (n * fact(n – 1));
}
```

Example 1 :: Factorial Execution




```
long  int  fact ( int n)
{
   if  (n = = 1) return (1);
   else return (n * fact(n – 1) );
}
```

Example 2 :: Fibonacci number

Fibonacci number f(n) can be defined as:

$f(0) = 0$

$f(1) = 1$

$f(n) = f(n-1) + f(n-2), \quad \text{if } n > 1$

- The successive Fibonacci numbers are:

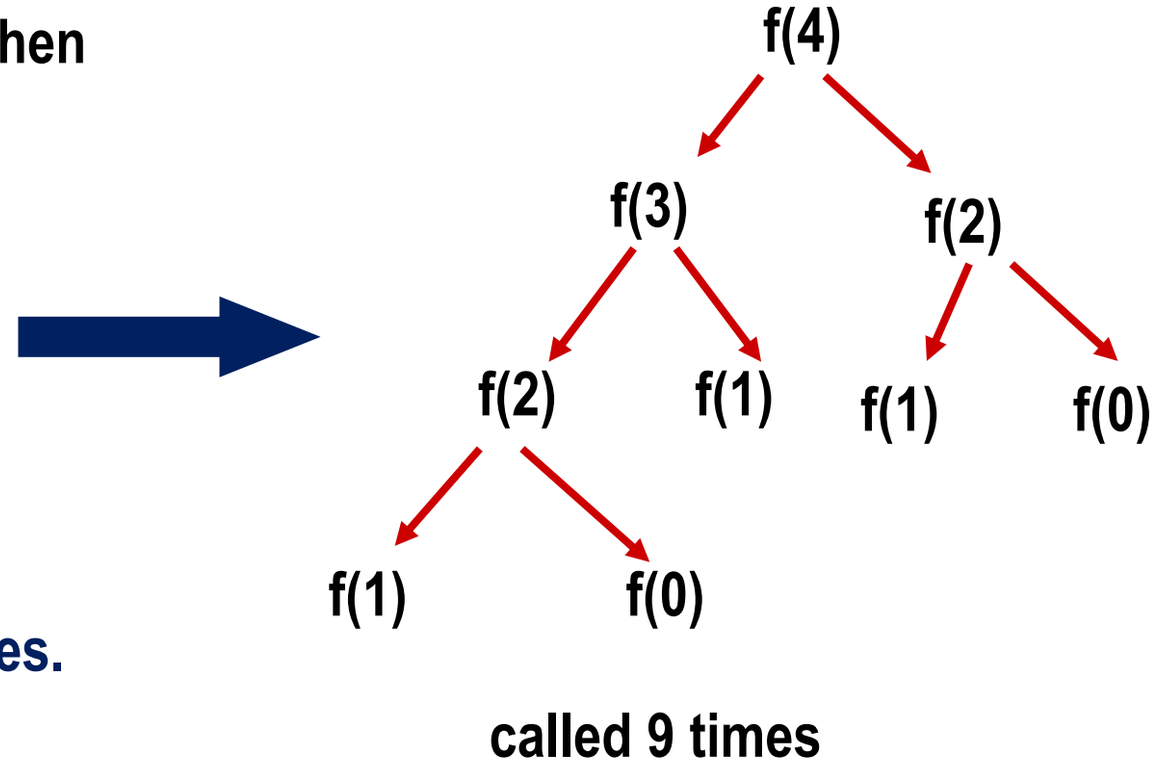
  0, 1, 1, 2, 3, 5, 8, 13, 21, …..

Function definition:

```
int   f (int n)
{
    if  (n  < 2)   return (n);
    else  return ( f(n – 1) + f(n – 2) );
}
```

Tracing Execution

How many times is the function called when evaluating f(4) ?



called 9 times

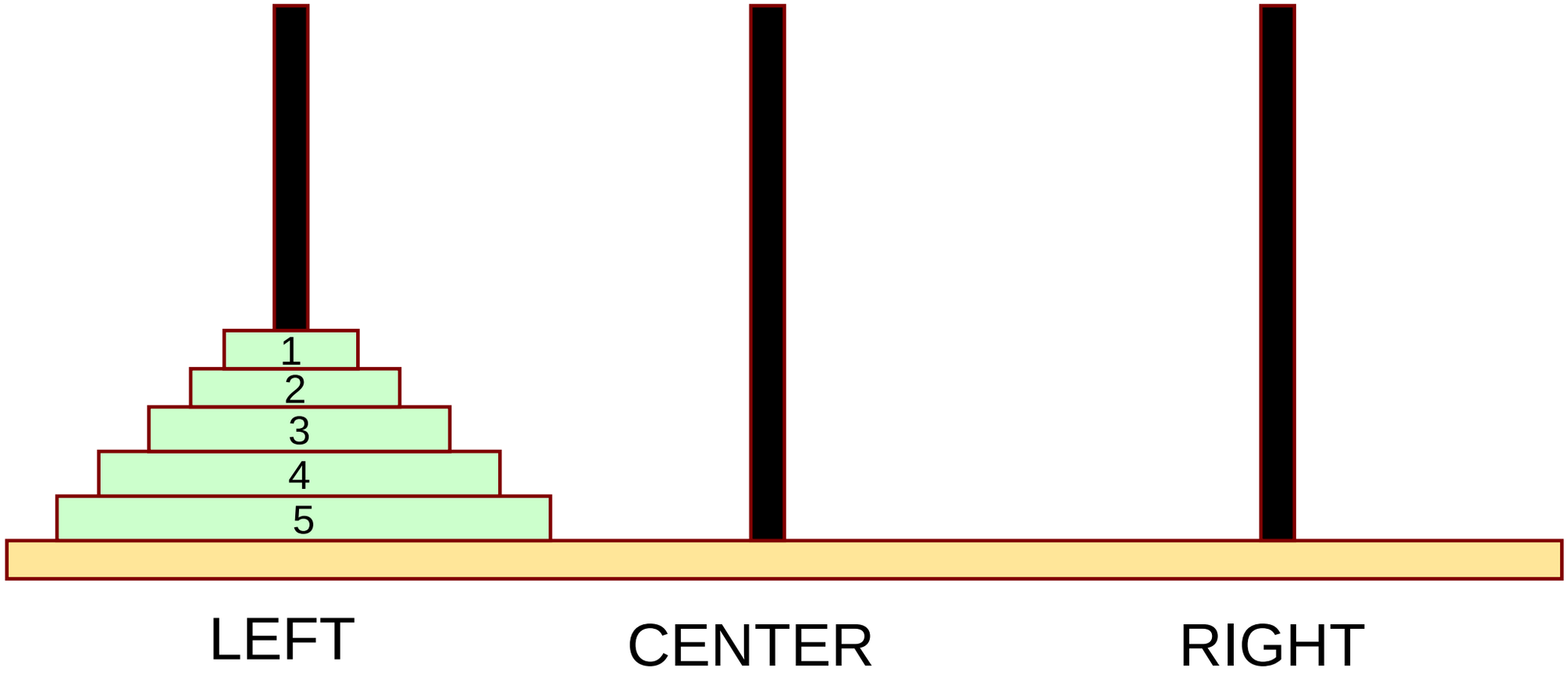
Inefficiency:

- **Same thing is computed several times.**

Notable Point

- Every recursive program can also be written without recursion
- Recursion is used for programming convenience, not for performance enhancement
- Sometimes, if the function being computed has a nice recurrence form, then a recursive code may be more readable

Example 3 :: Towers of Hanoi Problem



Towers of Hanoi

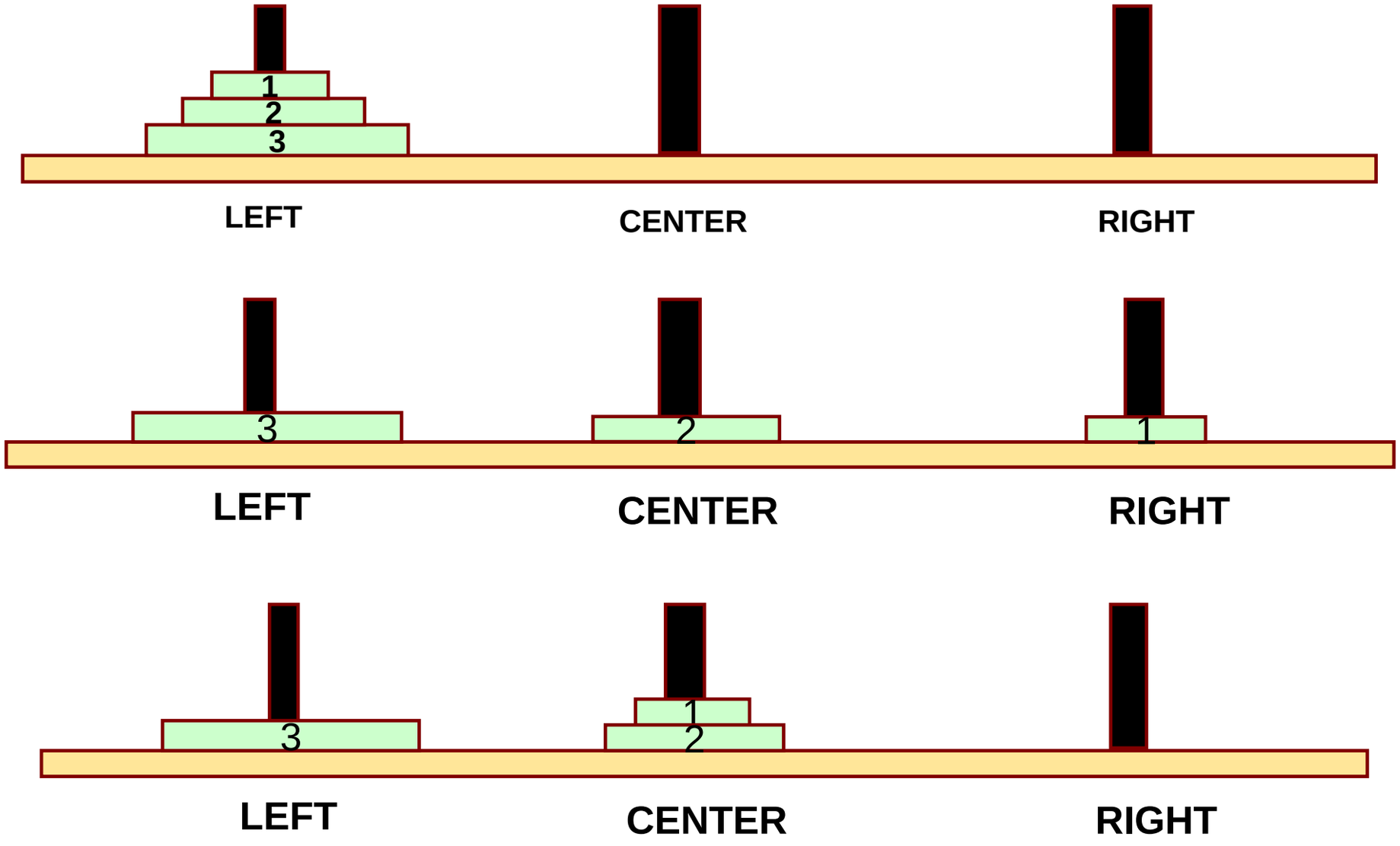
The problem statement:

- Initially all the disks are stacked on the LEFT pole.
- Required to transfer all the disks to the RIGHT pole.
  - Only one disk can be moved at a time.
  - A larger disk cannot be placed on a smaller disk.
- CENTER pole is used for temporary storage of disks.

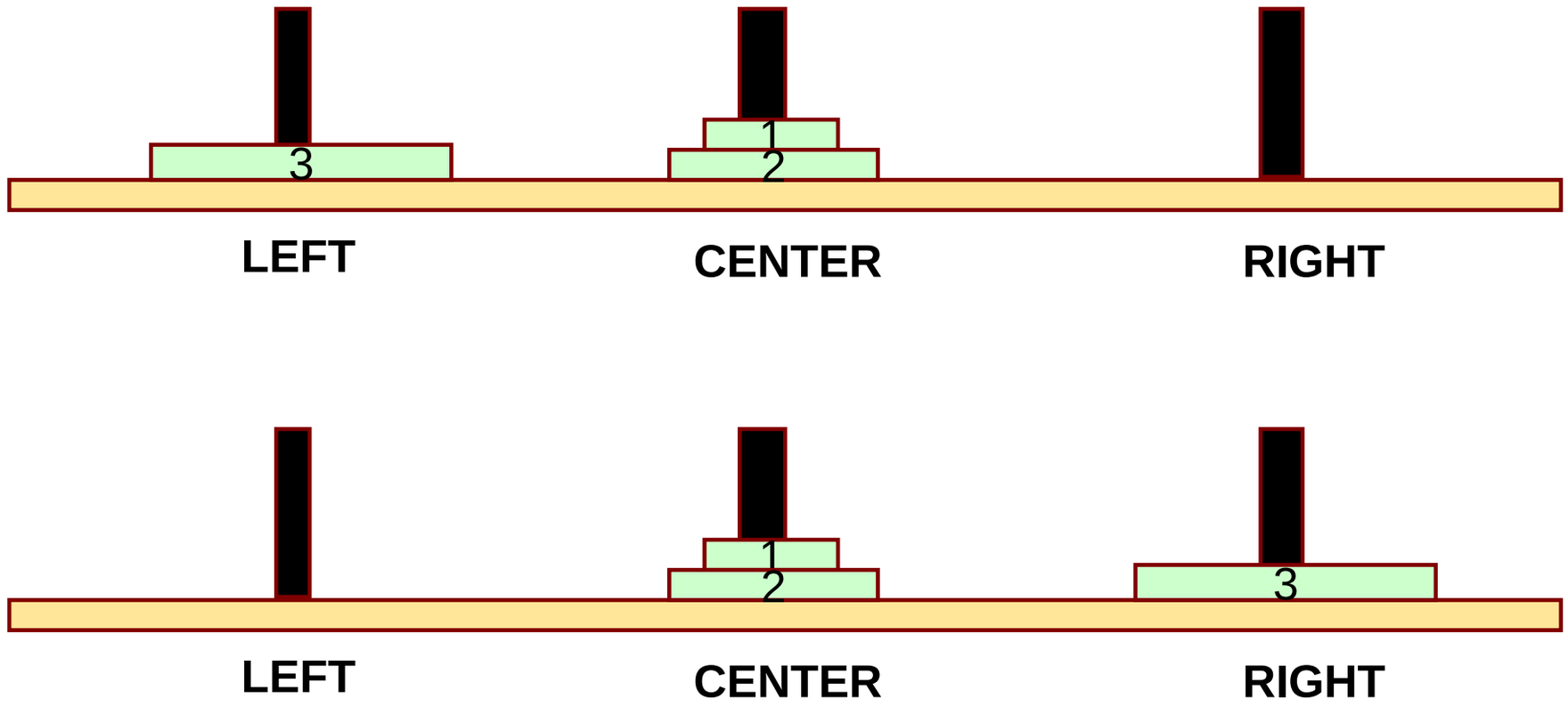
Recursive statement of the general problem of n disks.

- **Step 1:**
  - **Move the top (n-1) disks from LEFT to CENTER.**
- **Step 2:**
  - **Move the largest disk from LEFT to RIGHT.**
- **Step 3:**
  - **Move the (n-1) disks from CENTER to RIGHT.**

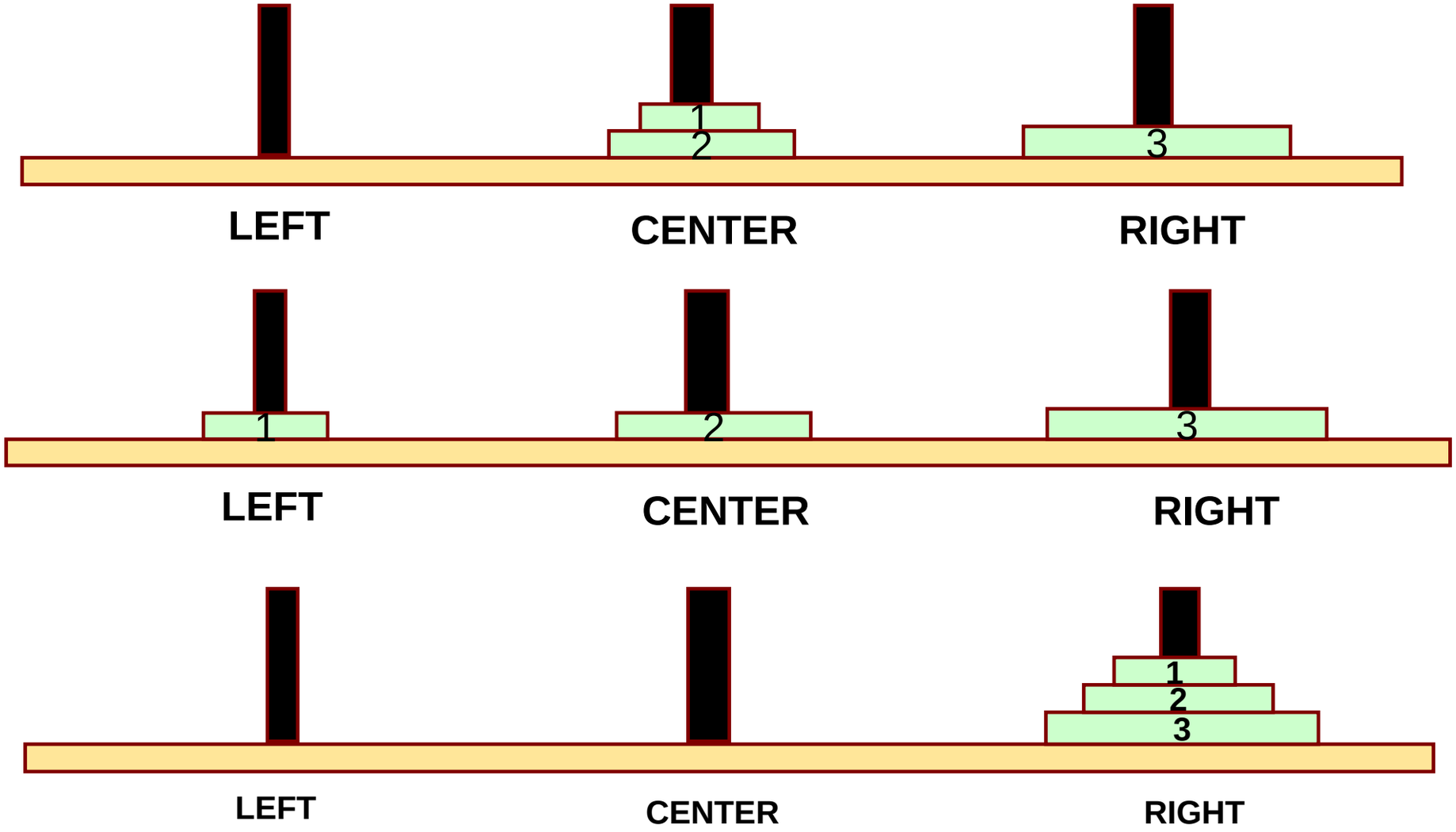
Phase-1: Move top n – 1 from LEFT to CENTER



Phase-2: Move the n^{th} disk from LEFT to RIGHT



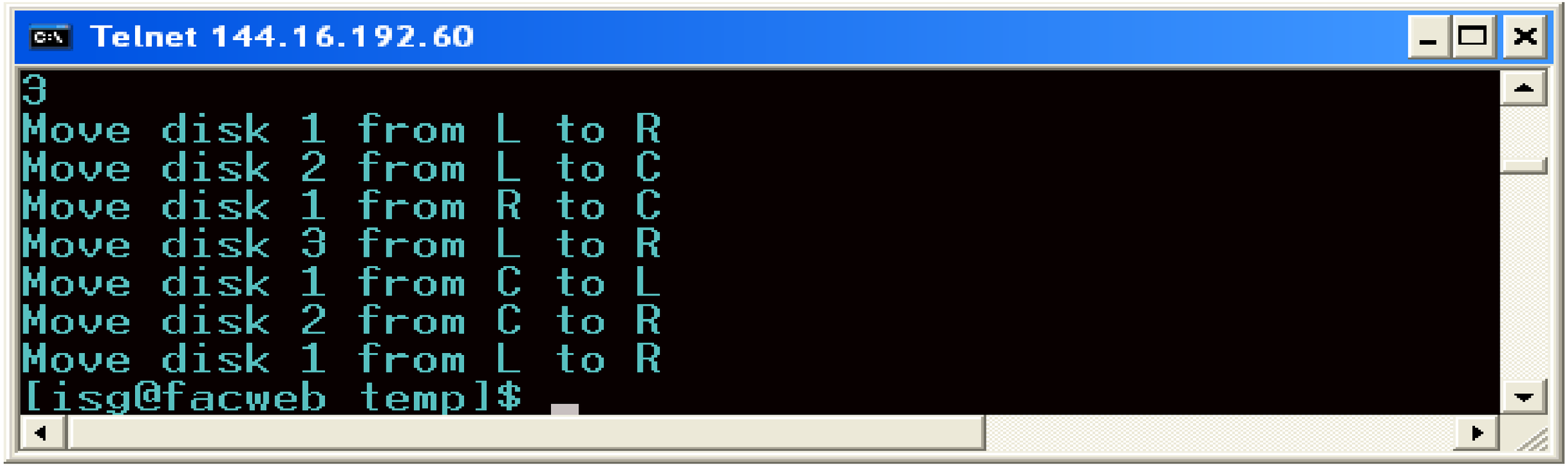
Phase-3: Move top n – 1 from CENTER to RIGHT

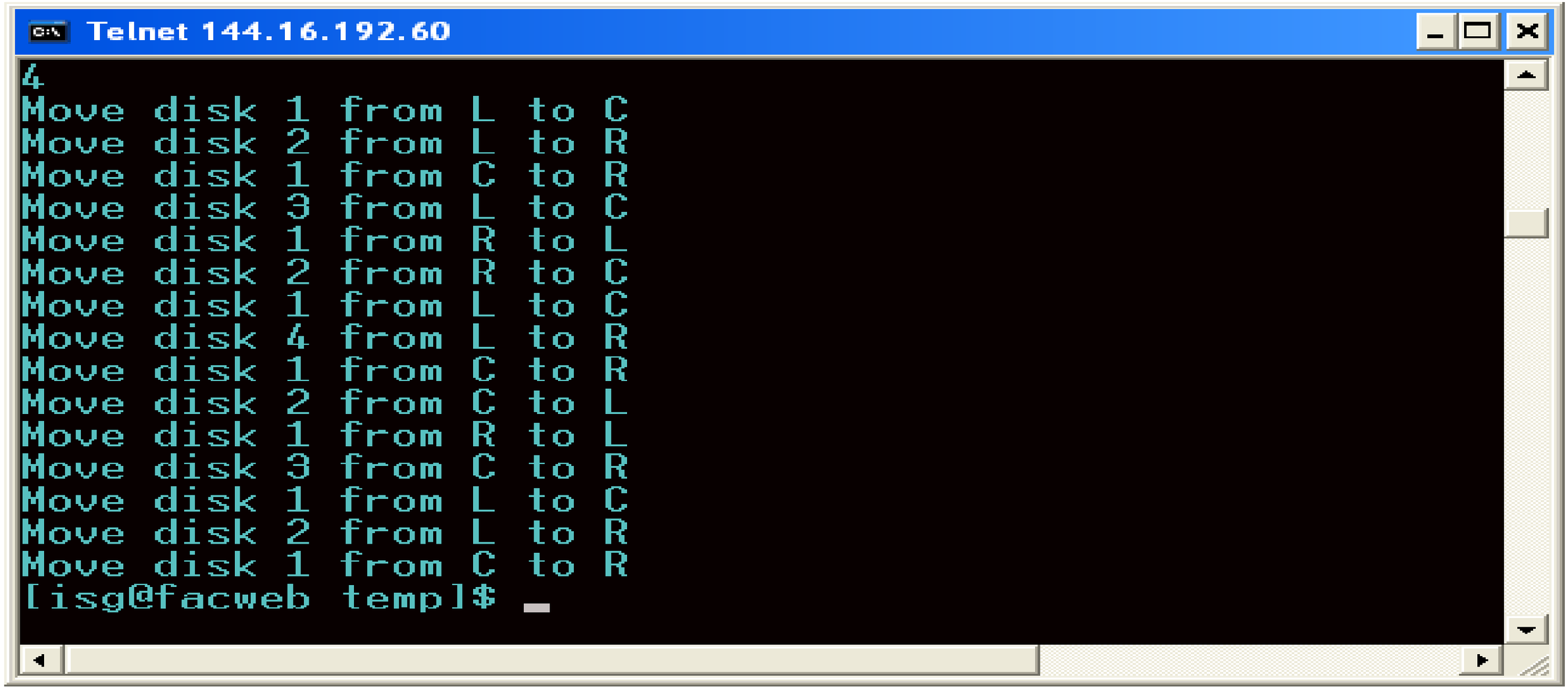


```
#include  <stdio.h>
void  transfer (int n, char from, char to, char temp);

main( )
{           int  n;  /* Number of disks */
            scanf (“%d”, &n);
            transfer (n, ‘L’, ‘R’, ‘C’);
}

void  transfer (int n, char from, char to, char temp)
{
            if  (n > 0)  {
                        transfer  (n-1, from, temp, to);
                        printf (“Move disk %d from %c to %c \n”, n, from, to);
                        transfer (n-1, temp, to, from);
            }
            return;
}
```


Telnet 144.16.192.60
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
[isg@facweb temp]$


Telnet 144.16.192.60
4
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
Move disk 3 from L to C
Move disk 1 from R to L
Move disk 2 from R to C
Move disk 1 from L to C
Move disk 4 from L to R
Move disk 1 from C to R
Move disk 2 from C to L
Move disk 1 from R to L
Move disk 3 from C to R
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
[isg@facweb temp]$

Recursion versus Iteration

Repetition

- Iteration: explicit loop
- Recursion: repeated function calls

Termination

- Iteration: loop condition fails
- Recursion: base case recognized

Both can have infinite loops

Balance

- Choice between performance (iteration) and good software engineering (recursion).

How are recursive calls implemented?

What we have seen ….

- Activation record gets pushed into the stack when a function call is made.
- Activation record is popped off the stack when the function returns.

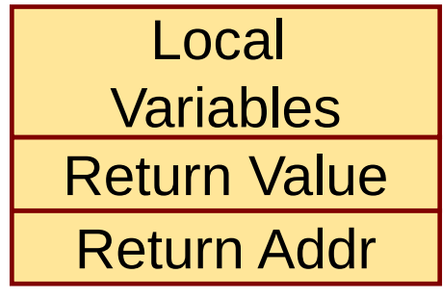
In recursion, a function calls itself.

- Several function calls going on, with none of the function calls returning back.
  - Activation records are pushed onto the stack continuously.
  - Large stack space required.

- **Activation records keep popping off, when the termination condition of recursion is reached.**

We shall illustrate the process by an example of computing factorial.

- **Activation record looks like:**

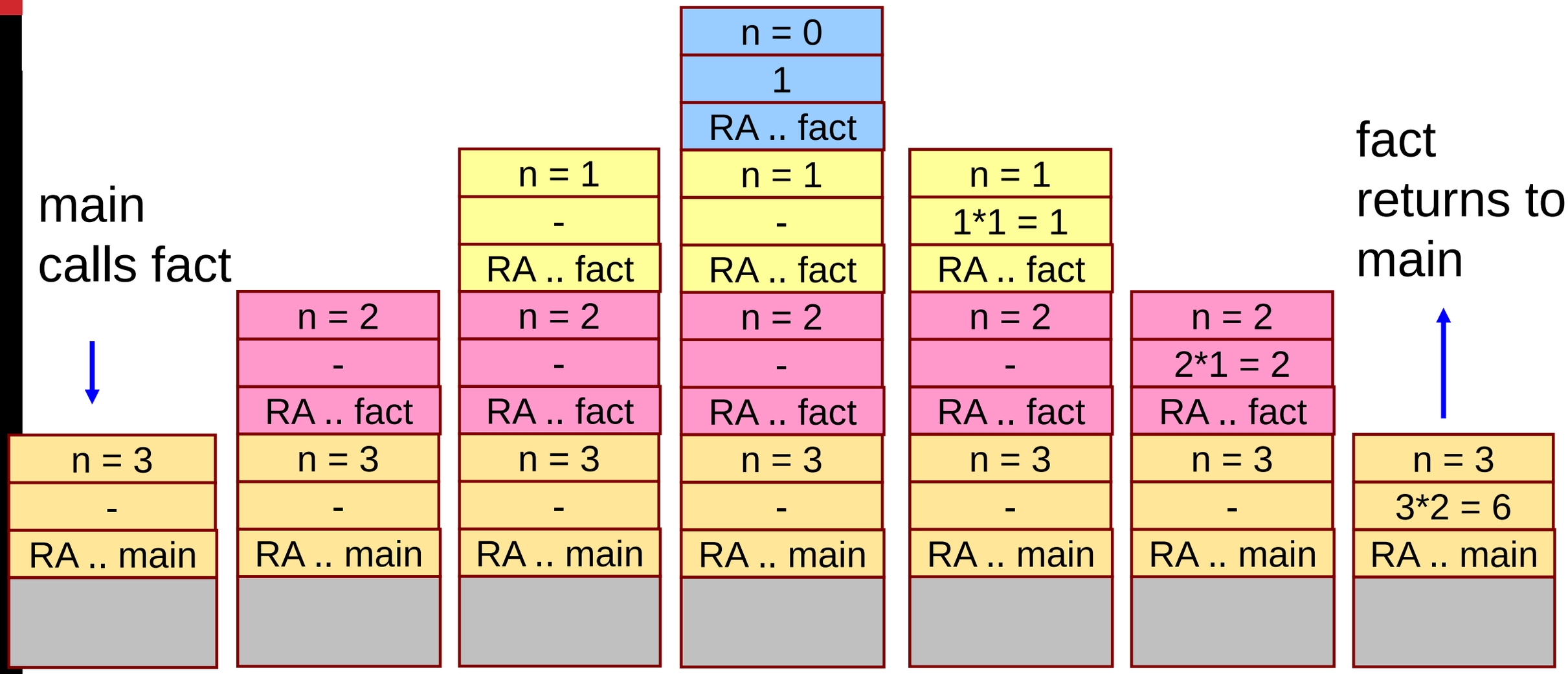


Example:: main( ) calls fact(3)

```
main()
{
   int  n;
   n = 3;
   printf (“%d \n”, fact(n) );
}
```

```
int  fact (n)
int  n;
{
        if   (n = = 0)
                return (1);
        else
                return  (n * fact(n-1));
}
```

TRACE OF THE STACK DURING EXECUTION



Do Yourself

Trace the activation records for the following version of Fibonacci sequence.

```
#include <stdio.h>
int   f (int n)
{
    int a, b;
    if  (n  < 2)   return (n);
    else  {
      a = f(n-1);
      b = f(n-2);
      return (a+b);  }
}

main( ) {
   printf("Fib(4) is: %d \n", f(4));
}
```

X

Y

main

Local Variables (n, a, b)
Return Value
Return Addr (either main, or X, or Y)

Examples

What do the following programs print?

```
void foo( int n )
{
    int data;
    if ( n == 0 ) return;
    scanf("%d", &data);
    foo ( n – 1 );
    printf("%d\n", data);
}
main ( )
{   int k = 5;
    foo ( k );
}
```

```
void foo( int n )
{
    int data;
    if ( n == 0 ) return;
    foo ( n – 1 );
    scanf("%d", &data);
    printf("%d\n", data);
}
main ( )
{   int k = 5;
    foo ( k );
}
```

```
void foo( int n )
{
    int data;
    if ( n == 0 ) return;
    scanf("%d", &data);
    printf("%d\n", data);
    foo ( n – 1 );
}
main ( )
{   int k = 5;
    foo ( k );
}
```

Printing cumulative sum -- *will this work?*

```
int foo(  int n  )
{
      int data, sum ;
      if ( n == 0 ) return 0;
      scanf("%d", &data);
      sum = data + foo ( n – 1 );
      printf("%d\n", sum);
      return sum;
}
main ( ) {
      int k = 5;
      foo ( k );
}
```

Input: 1 2 3 4 5

Output: 5 9 12 14 15

How to rewrite this so that the output is: 1 3 6 10 15 ?

Printing cumulative sum (two ways)

```
int foo(  int n  )
{
      int data, sum ;
      if ( n == 0 ) return 0;
      sum = foo ( n – 1 );
      scanf("%d", &data);
      sum = sum + data;
      printf("%d\n", sum);
      return sum;
}
main ( ) {
      int k = 5;
      foo ( k );
}
```

Input: 1 2 3 4 5

Output: 1 3 6 10 15

```
void foo(  int n, int sum  )
{
      int data ;
      if ( n == 0 ) return 0;
      scanf("%d", &data);
      sum = sum + data;
      printf("%d\n", sum);
      foo( k – 1, sum ) ;
}
main ( ) {
      int k = 5;
      foo ( k, 0 );
}
```

What does this program print?

```
#include <stdio.h>

int factorial (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}
```

```
main()
{
    int i=6;
    printf ("Value is: %d \n", factorial(i));
}
```

What does this program print?

```
#include <stdio.h>
int factorial (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n == 0) return 1;
    else return (n * factorial(n-1));
}

main()
{    int i=6;
     printf ("Value is: %d \n", factorial(i));
}
```

Program output:

```
n=6, count=1
n=5, count=2
n=4, count=3
n=3, count=4
n=2, count=5
n=1, count=6
n=0, count=7
Value is: 720
```

What does this program print?

```
#include <stdio.h>
int fib (int n)
{
    static int count=0;
    count++;
    printf (“n=%d, count=%d \n”, n, count);
    if (n < 2) return n;
            else return (fib(n-1) + fib(n-2));
}
main( )
{    int i=4;
     printf (“Value is: %d \n”, fib(i));
}
```

What does this program print?

```
#include <stdio.h>
int fib (int n)
{
    static int count=0;
    count++;
    printf ("n=%d, count=%d \n", n, count);
    if (n < 2) return n;
            else return (fib(n-1) + fib(n-2));
}
main( )
{    int i=4;
     printf ("Value is: %d \n", fib(i));
}
```

n=4, count=1

n=3, count=2

n=2, count=3

n=1, count=4

n=0, count=5

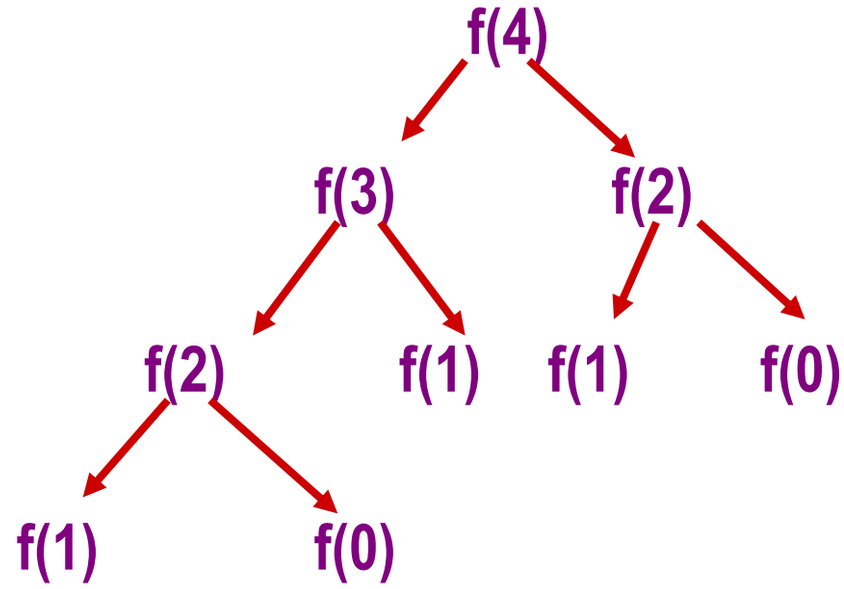
n=1, count=6

n=2, count=7

n=1, count=8

n=0, count=9

Value is: 3 [0,1,1,2,3,5,8,….]



Merge-Sort

```
void mergesort ( int a[ ], int lo, int hi )
{
        int m;
        if (lo<hi) {
                m=(lo+hi)/2;
                mergesort(a, lo, m);
                mergesort(a, m+1, hi);
                merge(a, lo, m, hi);
        }
}
```

Function Merge

```
void merge ( int a[ ], int lo, int m, int hi )
{
        int i, j, k;

        // copy both halves of a to auxiliary array b
        for (i=lo; i<=hi; i++) b[i]=a[i];

        i=lo; j=m+1; k=lo;
        // copy back next-greatest element at each time
        while (i<=m && j<=hi)
                if (b[i]<=b[j]) a[k++]=b[i++];
                else a[k++]=b[j++];

        // copy back remaining elements of first half (if any)
        while (i<=m) a[k++]=b[i++];
}
```

Recursive Permutation Generator

```
#define SWAP(x, y, t) { (t) == (x); (x) = (y); (y) = (t) }
void perm (char list[ ], int i, int n)
{
        int j, tmp;
        if (i == n) {
                for (j=0; j<=n; j++) printf("%c", list[ j ]);
                printf("\n");
        }
        else {
                for (j=i; j <= n; j++) {
                        SWAP(list[ i ], list[ j ], tmp);
                        perm(list, i+1, n);
                        SWAP(list[ i ], list[ j ], tmp);
                }
        }
}
```

Transitive Closure

```
Transclosure ( int adjmat[ ][max], int path[ ][max] )
{
        for (i = 0; i < max; i++)
                for (j = 0; j < max; j++)
                        path[i][j] = adjmat[i][j];

        for (k = 0; k < max; k++)
            for (i = 0; i < max; i++)
                for (j = 0; j < max; j++)
                    if ((path[i][k] == 1)&&(path[k][j] == 1)) path[i][j] = 1;
}
```

Paying with fewest coins

- A country has coins of denomination 3, 5 and 10 respectively.
- We are to write a function canchange(k) that returns –1 if it is not possible to pay a value of k using these coins.
  - Otherwise it returns the minimum number of coins needed to make the payment.

- For example, canchange(7) will return –1.
- On the other hand, canchange(14) will return 4 because 14 can be paid as 3+3+3+5 and there is no other way to pay with fewer coins.

Paying with fewest coins

```
int canchange( int k )
{
        int a = –1;
        if (k==0) return 0;
        if ( _____ ) return 1;
        if (k < 3)  _____ ;

        a = canchange( _____ ); if (a > 0) return _____ ;
        a = canchange(k – 5); if (a > 0) return _____ ;
        a = canchange( _____ ); if (a > 0) return _____ ;
        return –1;
}
```

Paying with fewest coins

```
int canchange( int k )
{
        int a = –1;
        if (k==0) return 0;
        if ( (k ==3) || (k == 5) || (k == 10) ) return 1;
        if (k < 3)  return –1  ;

        a = canchange( k – 10 ); if (a > 0) return a+1 ;
        a = canchange( k – 5 ); if (a > 0) return a+1 ;
        a = canchange( k – 3 ); if (a > 0) return a+1 ;
        return –1;
 }
```