RECURSION

CS10003: PROGRAMMING AND DATA STRUCTURES

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 (assume that m and n are non-negative and m ≥ n):**

  gcd (m, 0) = m

  gcd (m, n) = gcd (n, m%n) , if n > 0

- **Fibonacci sequence (0,1,1,2,3,5,8,13,21,…)**

  fib (0) = 0

  fib (1) = 1

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

Example 1 :: Factorial

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

Example 1 :: Factorial Execution

```
fact(4)                                       24
  if   (4 = = 1) return (1);
  else return  (4 * fact(3));                            6
      if   (3 = = 1) return (1);
      else return  (3 * fact(2));                               2
          if   (2 = = 1) return (1);
          else return  (2 * fact(1));                                 1
              if   (1 = = 1) return (1);
              else return  (1 * fact(0));
```

```
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, …..

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

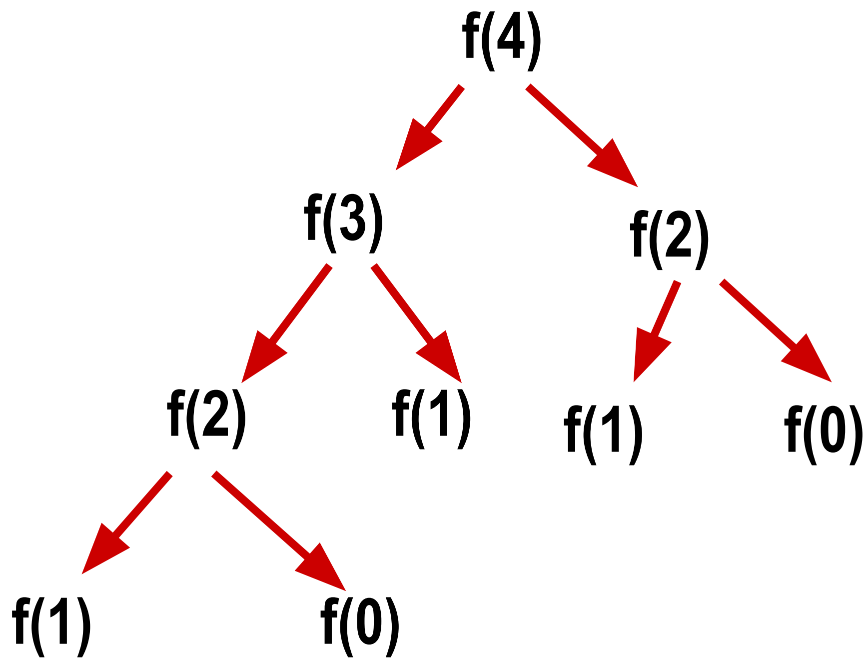
Tracing Execution

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

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

Inefficiency:

- Same thing is computed several times.



called 9 times

Some points to note

Every recursive program can also be written without recursion

- Tail Recursion: Last thing a recursive function does is making a single recursive call (of itself) at the end.
- Easy to replace tail recursion by a loop.
- In general, removal of recursion may be a very difficult task (even if you have your own recursion stack).

Recursion can be helpful in many situations

- Better readability
- Ease of programming
- Sometimes, recursion gives best-possible or best-known algorithms to solve problems

Recursion can also be a killer

- You solve the same subproblem multiple times (Example: Fibonacci numbers)
- Every recursive call incurs a (small) overhead

Use recursion with caution

Example of tail recursion

Not a tail recursion:

```
int sum1 ( int n )
{
        if (n == 0) return 0;
        return n + sum1(n–1);
}
```

Tail recursion:

```
int sum2 ( int n, int partialsum )
{
        if (n == 0) return partialsum;
        return sum2(n – 1, n + partialsum);
}
```

Call from main() as:

```
scanf("%d", &N);
s = sum2(N, 0);
```

Equivalent iterative function:

```
int sum3 ( int n )
{
        int partialsum = 0;
        while (n > 0) {
                partialsum = n + partialsum;
                n = n – 1;
        }
        return partialsum;
}
```

Important things to remember

- Think how the current problem can be solved if you can solve exactly the same problem on one or more smaller instance(s).
- Do NOT think how the problem will be solved on smaller instances, just call the function recursively and assume that the recursive calls do their jobs correctly.
- Do NOT forget to include the base cases to solve the problem on *smallest* instances.
- This is basically mathematical induction applied to programming.

- When you write a recursive function
    - First, write the terminating/base condition
    - Then, write the rest of the function
    - Always double-check that you have both

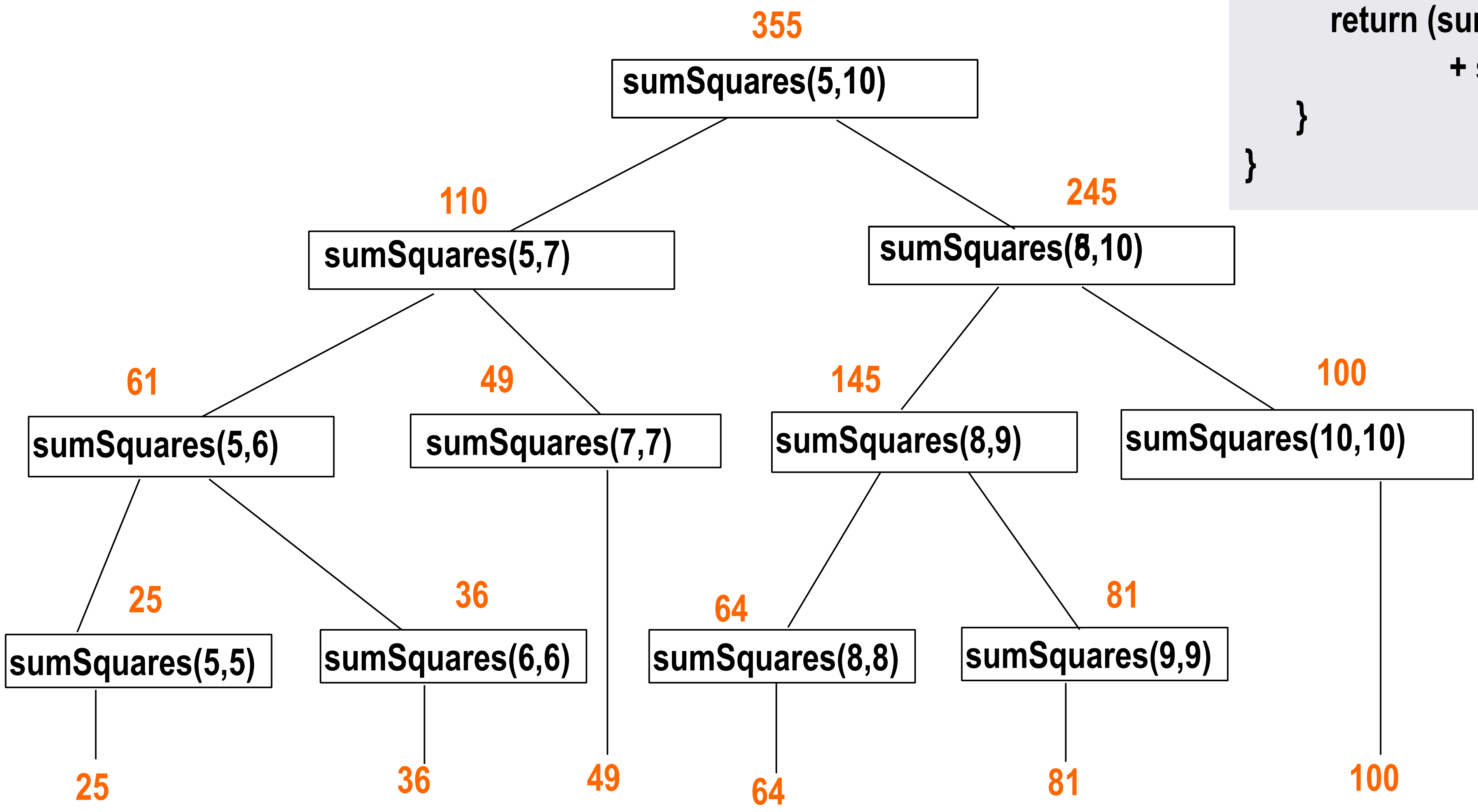
Example: Sum of Squares

Write a function that takes two integers m and n as arguments, and computes and returns the sum of squares of every integer in the range [m:n], both inclusive.

```
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n) return(m*m);
    else
    {
        middle = (m+n)/2;
        return (sumSquares(m,middle)  + sumSquares(middle+1,n));
     }
}
```

Annotated Call Tree

```
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n) return(m*m);
    else {
        middle = (m+n)/2;
        return (sumSquares(m,middle)
                + sumSquares(middle+1,n));
    }
}
```



Example: Printing the digits of an integer in reverse

Print the last digit, then print the remaining number in reverse

- Ex: If integer is 743, then reversed is print 3 first, then print the reverse of 74

```
void printReversed( int i )
{
    if (i < 10)   {
       printf("%d\n", i); return;
    }
    else {
        printf("%d", i%10);
        printReversed(i/10);
    }
}
```

Example: Printing your name in reverse

```
#include <stdio.h>

void readandprint ()
{
        char c;

        scanf("%c", &c);
        if (c == '\n') return;
        readandprint();
        printf("%c", c);
}

int main ()
{
        printf("Enter your name and hit return: ");
        readandprint();
        printf("\n");
}
```

Output

```
Enter your name and hit return: Jane Doe
eoD enaJ
```

Exercise: Rewrite this code so that the output looks as follows:

```
Enter your name and hit return: Jane Doe
Your name in reverse: eoD enaJ
```

Counting Zeros in a Positive Integer

Check last digit from right

- If it is 0, number of zeros = 1 + number of zeroes in remaining part of the number
- If it is non-0, number of zeros = number of zeroes in remaining part of the number

```
int zeros(int number)
{
    if(number < 10) return 0;
    if (number % 10 == 0)
        return( 1 + zeros(number/10) );
    else
      return( zeros(number/10) );
}
```

Common Errors in Writing Recursive Functions

Non-terminating Recursive Function (Infinite recursion)

- No base case

```
int badFactorial(int x) {
   return x * badFactorial(x-1);
}
```

- The base case is never reached

```
int badSum2(int x)
{
   if(x==1) return 1;
   return(badSum2(x--));
}
```

```
int anotherBadFactorial(int x) {
   if(x == 0)
      return 1;
   else
      return x*(x-1)*anotherBadFactorial(x-2);
      // When x is odd, base case is never reached!!
}
```

Common Errors in Writing Recursive Functions

Mixing up loops and recursion

```
int anotherBadFactorial(int x) {
        int i, fact = 0;
        if (x == 0) return 1;
        else {
                for (i=x; i>0; i=i-1) {
                        fact = fact + x*anotherBadFactorial(x-1);
                }
                return fact;
        }
}
```

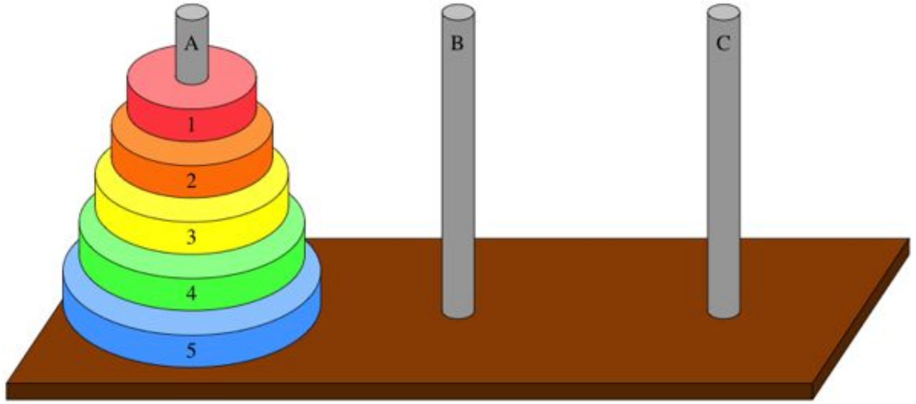
In general, if you have recursive function calls within a loop, think carefully if you need it.

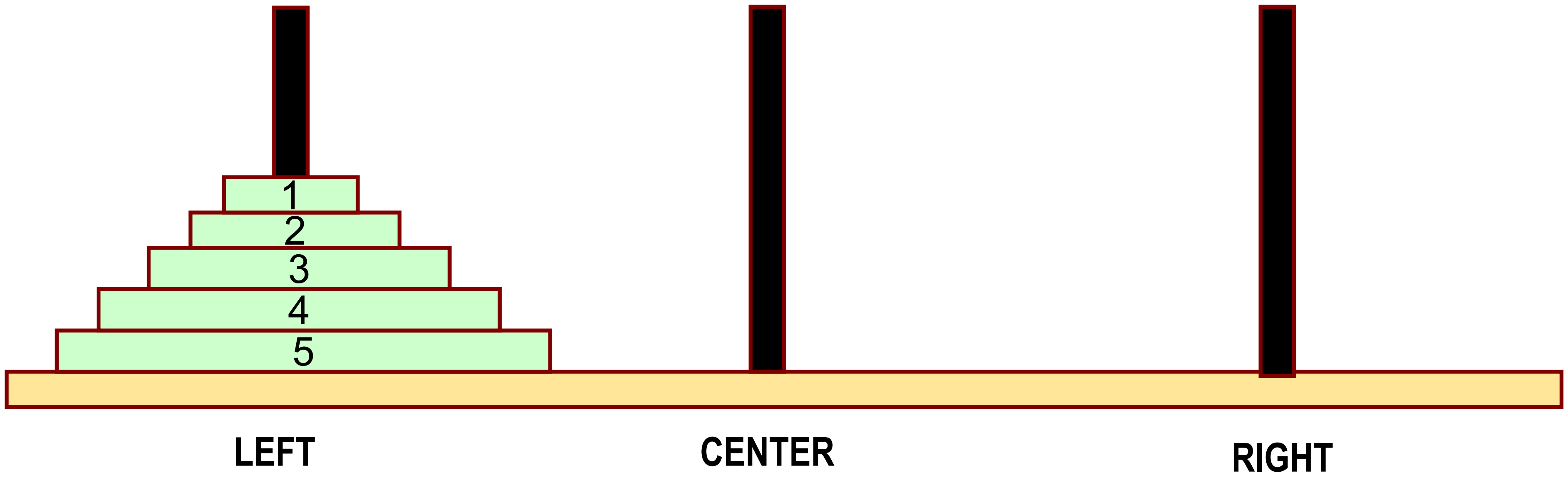
Most recursive functions you will see in this course will not need this

Example :: Towers of Hanoi Problem

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 on the top 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.

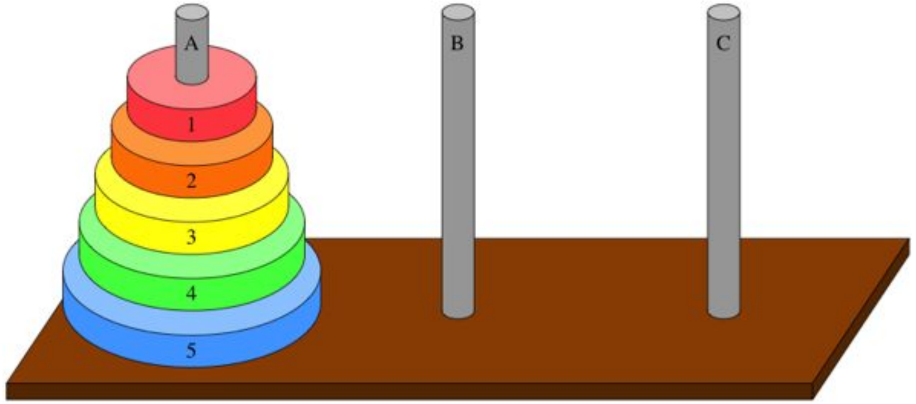




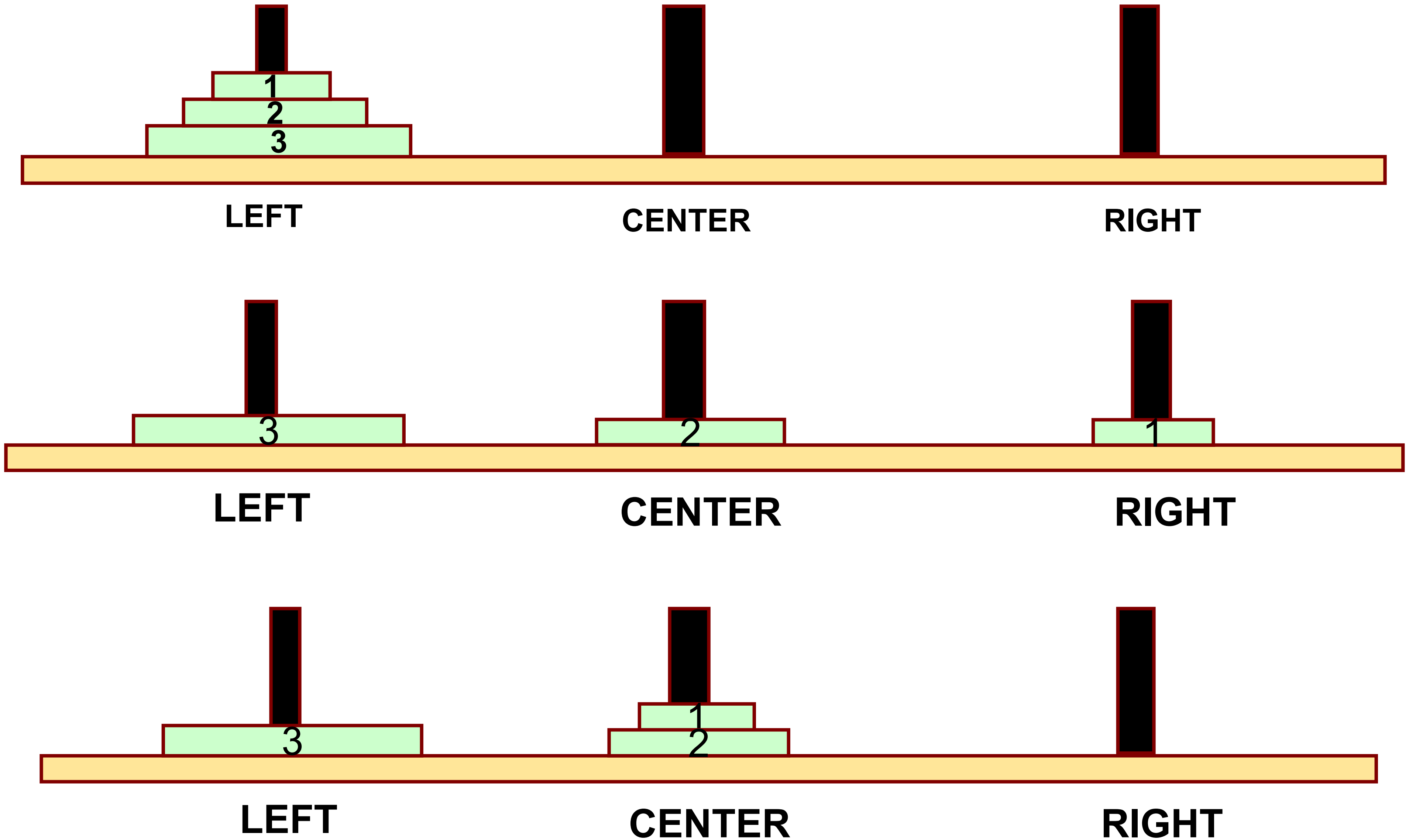
Recursive Formulation

Recursive statement of the general problem of n disks.

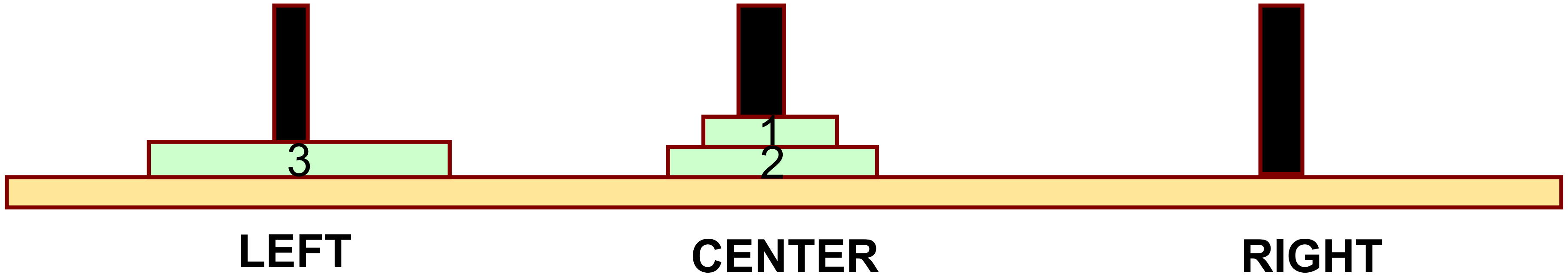
- **Step 1:**
  - **Move the top (n-1) disks from LEFT to CENTER.**
- **Step 2:**
  - **Move the largest disk from LEFT to RIGHT.**
- **Step 3:**
  - **Move the (n-1) disks from CENTER to RIGHT.**

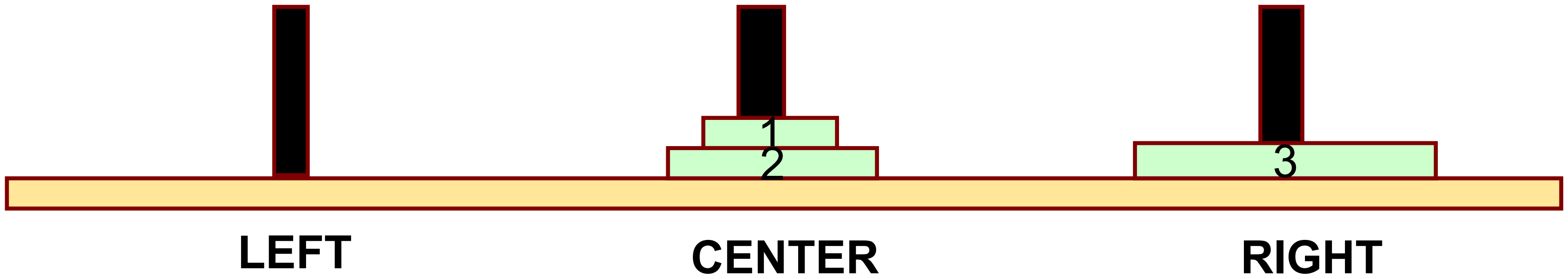


Phase-1: Move top n – 1 from LEFT to CENTER

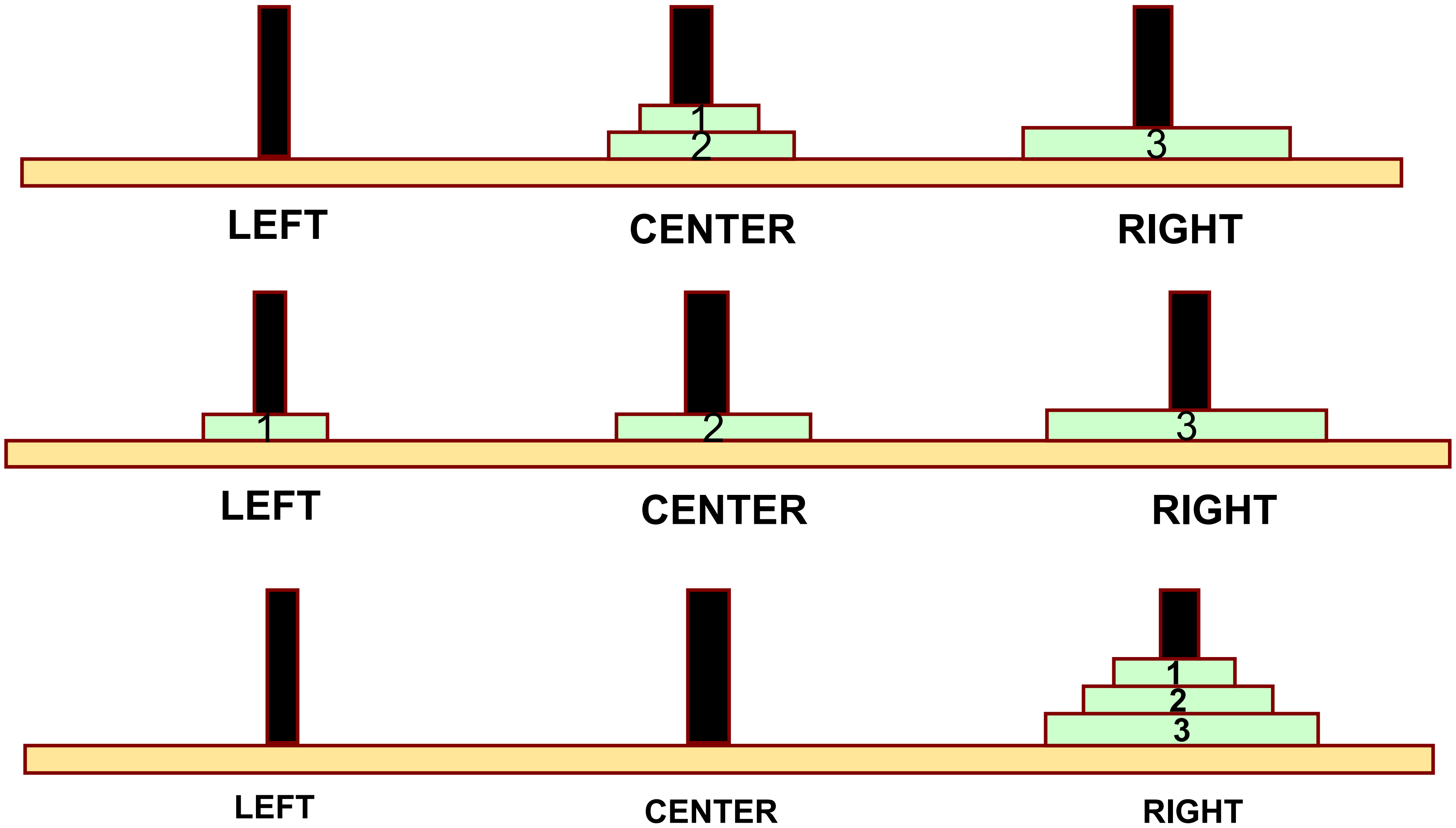


Phase-2: Move the n^{th} disk from LEFT to RIGHT





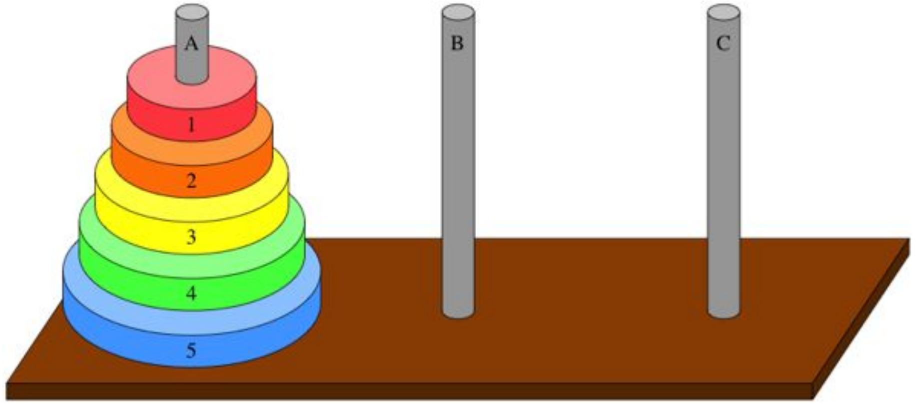
Phase-3: Move top n – 1 from CENTER to RIGHT



```
#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]$
```

With 3 discs

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]$
```

With 4 discs

Recursion versus Iteration

Repetition

- Iteration: explicit loop
- Recursion: repeated nested function calls

Termination

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

Both can have infinite loops

Balance

- Understand the benefits / penalties of recursion in terms of
  - Ease of implementation
  - Readability
  - Performance degradation / performance enhancement
- Take an educated decision

More 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 );
}
```

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.
- Finally, 15 can be changed as 3+3+3+3+3, 5+5+5, 5+10, so canchange(15) will return 2.

Paying with fewest coins

```
int canchange( int k )
{
    int a;
    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;
        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;
}
```

Exercise: Rewrite this code if the denominations are 3, 8, and 10. Do you see a problem? Repair it.

Practice Problems

1. Write a recursive function to search for an element in an array
2. Write a recursive function to count the digits of a positive integer (do also for sum of digits)
3. Write a recursive function to reverse a null-terminated string
4. Write a recursive function to convert a decimal number to binary
5. Write a recursive function to check if a string is a palindrome or not
6. Write a recursive function to copy one array to another

Note:

- For each of the above, write the main functions to call the recursive function also
- Practice problems are just for practicing recursion, recursion is not necessarily the most efficient way of doing them

Advanced topic

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:

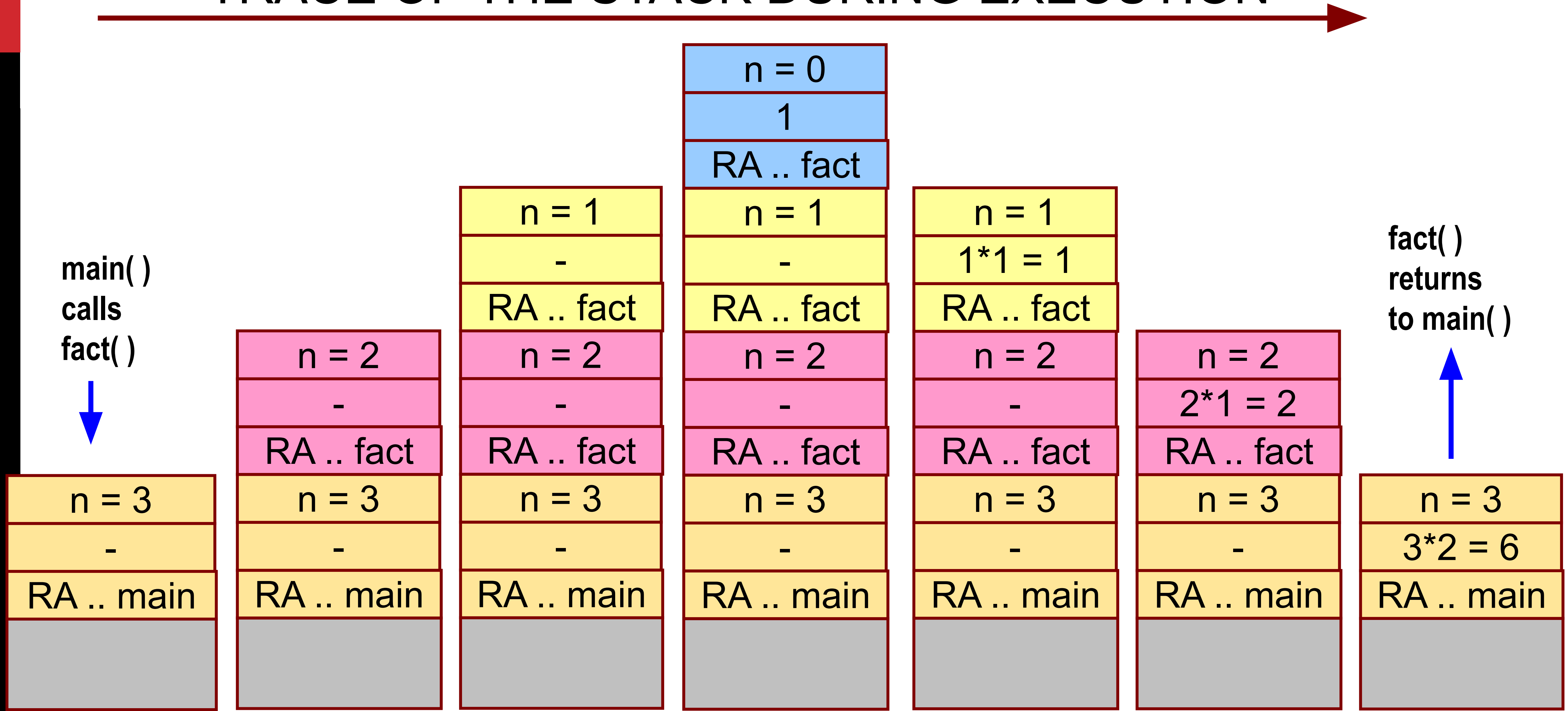
Actual Parameters
Local Variables
Return Value
Return Address
. . .

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  {
X  →   a = f(n-1);
Y  →   b = f(n-2);
       return (a+b);  }
}

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

Activation Record
Actual Parameters (n)
Local Variables (a, b)
Return Value
Return Address (either main or f)