



Recursion

Recursion

- A process by which a function calls itself repeatedly
 - Either directly.
 - X calls X
 - Or cyclically in a chain.
 - X calls Y, and Y calls X
- Used for repetitive computations in which each action is stated in terms of a previous result
$$\text{fact}(n) = n * \text{fact}(n-1)$$

Contd.

- 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
 - Solution of the problem in terms of solution of the same problem on smaller sized data
 - The problem statement must include a stopping condition

$$\begin{aligned} \text{fact}(n) &= 1, & \text{if } n = 0 & \leftarrow \text{Stopping condition} \\ &= n * \text{fact}(n-1), & \text{if } n > 0 & \leftarrow \text{Recursive definition} \end{aligned}$$

■ Examples:

□ Factorial:

$$\text{fact}(0) = 1$$

$$\text{fact}(n) = n * \text{fact}(n-1), \text{ if } n > 0$$

□ GCD:

$$\text{gcd}(m, m) = m$$

$$\text{gcd}(m, n) = \text{gcd}(m \% n, n), \text{ if } m > n$$

$$\text{gcd}(m, n) = \text{gcd}(n, n \% m), \text{ if } m < n$$

□ Fibonacci series (1, 1, 2, 3, 5, 8, 13, 21,

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

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

Factorial

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

Factorial Execution

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

Factorial Execution

fact(4)



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

Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```



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


Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```



```
if (3 == 1) return (1);  
else return (3 * fact(2));
```



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

Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```



```
if (3 == 1) return (1);  
else return (3 * fact(2));
```



```
if (2 == 1) return (1);  
else return (2 * fact(1));
```



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

Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```



```
if (3 == 1) return (1);  
else return (3 * fact(2));
```



```
if (2 == 1) return (1);  
else return (2 * fact(1));
```



```
if (1 == 1) return (1);
```

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

Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```

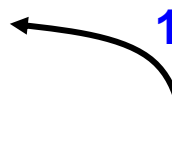


```
if (3 == 1) return (1);  
else return (3 * fact(2));
```



```
if (2 == 1) return (1);  
else return (2 * fact(1));
```

1



```
if (1 == 1) return (1);
```

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

Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```



```
if (3 == 1) return (1);  
else return (3 * fact(2));
```



```
if (2 == 1) return (1);  
else return (2 * fact(1));
```



```
if (1 == 1) return (1);
```

2

1

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

Factorial Execution

fact(4)



```
if (4 == 1) return (1);  
else return (4 * fact(3));
```



```
if (3 == 1) return (1);  
else return (3 * fact(2));
```



```
if (2 == 1) return (1);  
else return (2 * fact(1));
```



```
if (1 == 1) return (1);
```

2

1

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

Factorial Execution

fact(4)

↓
if (4 == 1) return (1);
else return (4 * fact(3));

↓
if (3 == 1) return (1);
else return (3 * fact(2));

↓
if (2 == 1) return (1);
else return (2 * fact(1));

↓
if (1 == 1) return (1);

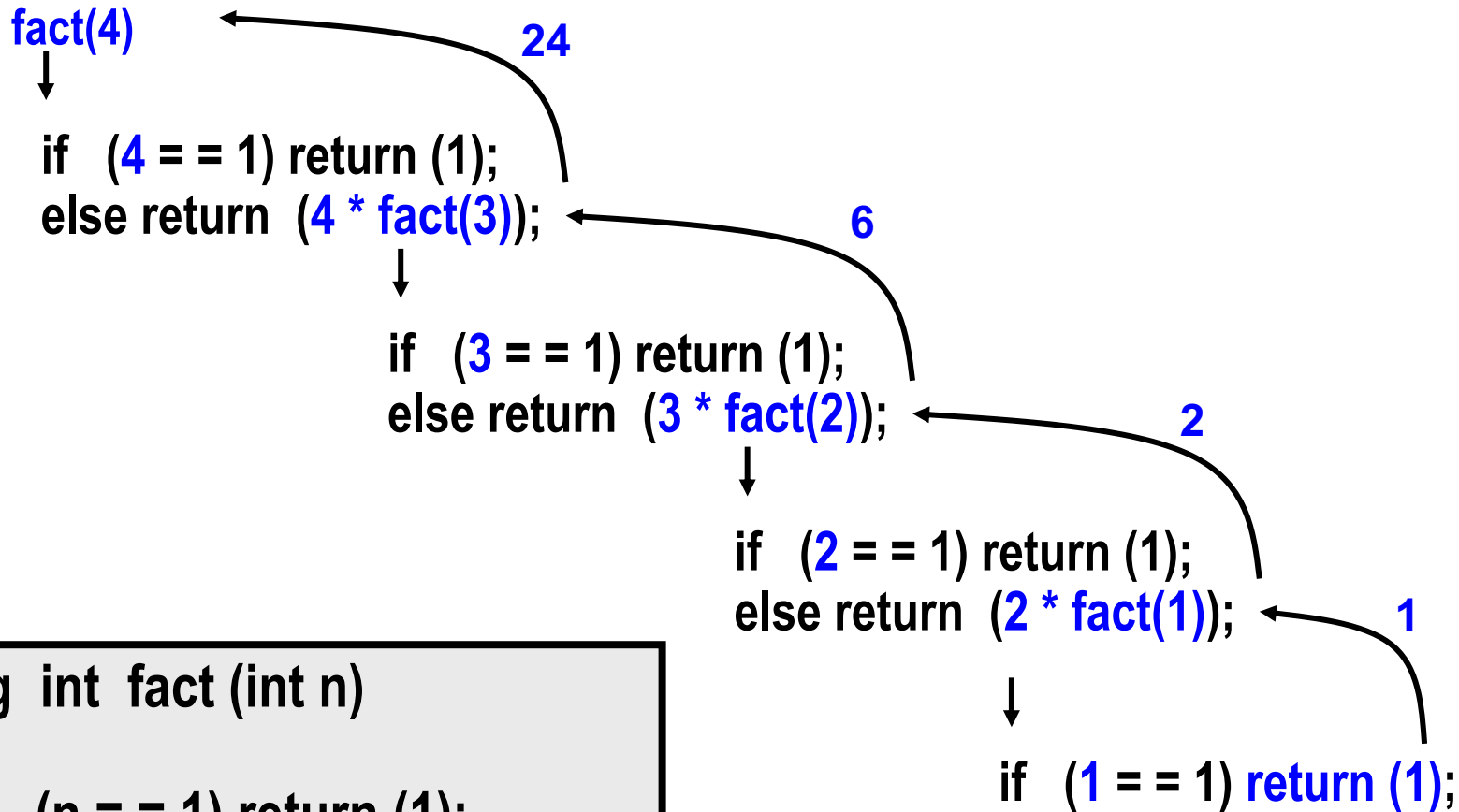
6

2

1

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

Factorial Execution



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


Look at the variable addresses (a slightly different program) !

```
void main()
{
    int x,y;
    scanf("%d",&x);
    y = fact(x);
    printf ("M: x= %d, y = %d\n", x,y);
}

int fact(int data)
{ int val = 1;
  printf("F: data = %d, &data = %u \n
    &val = %u\n", data, &data, &val);
  if (data>1) val = data*fact(data-1);
  return val;
}
```

Output

```
4
F: data = 4, &data = 3221224528
  &val = 3221224516
F: data = 3, &data = 3221224480
  &val = 3221224468
F: data = 2, &data = 3221224432
  &val = 3221224420
F: data = 1, &data = 3221224384
  &val = 3221224372
M: x= 4, y = 24
```

Fibonacci Numbers

Fibonacci recurrence:

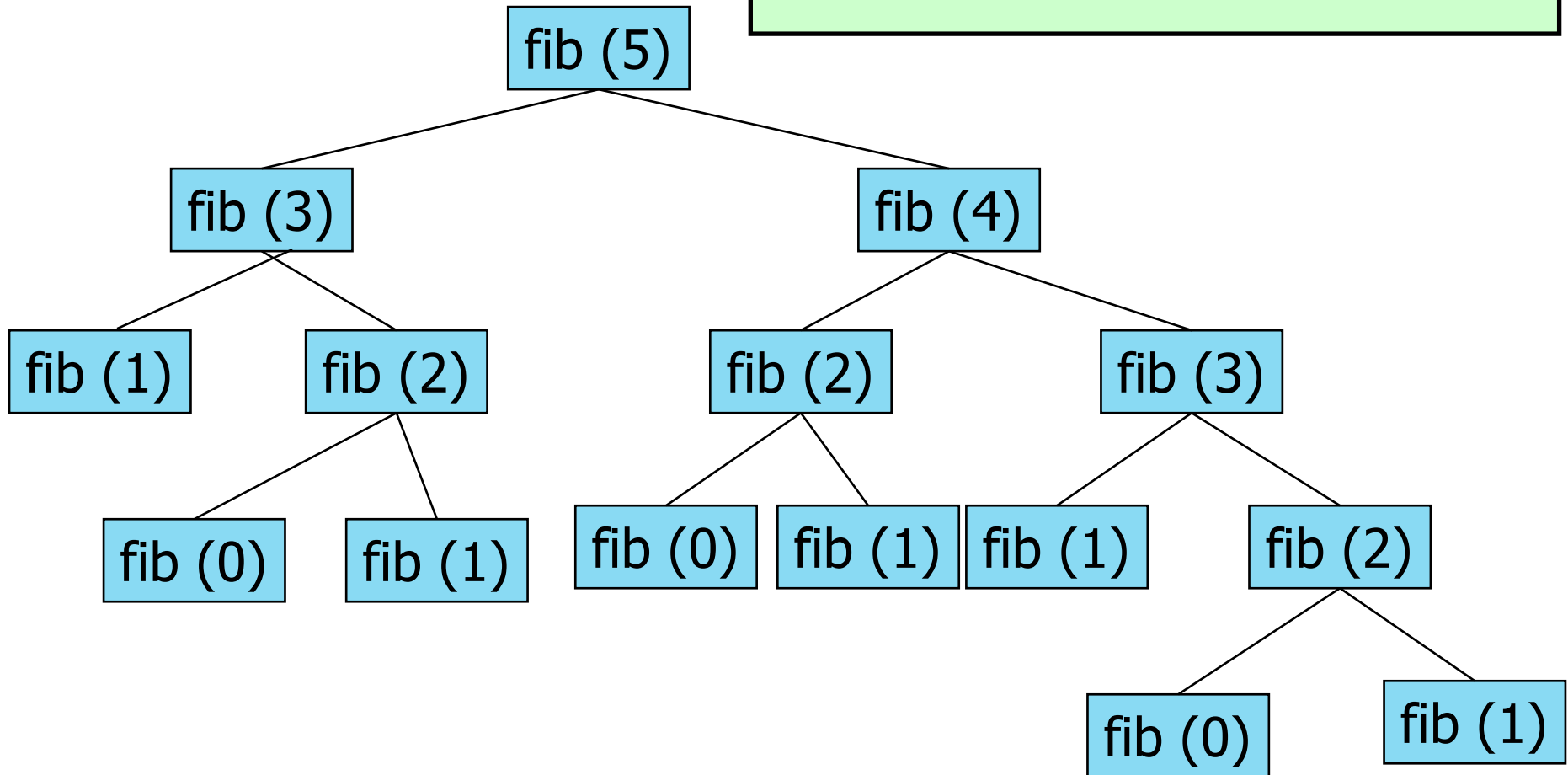
**fib(n) = 1 if n = 0 or 1;
 = fib(n - 2) + fib(n - 1)
 otherwise;**

```
int fib (int n){  
    if (n == 0 or n == 1)  
        return 1;    [BASE]  
    return fib(n-2) + fib(n-1) ;  
                    [Recursive]  
}
```

```
int fib (int n) {  
    if (n == 0 || n == 1)  
        return 1;  
    return fib(n-2) + fib(n-1);  
}
```

Fibonacci recurrence:

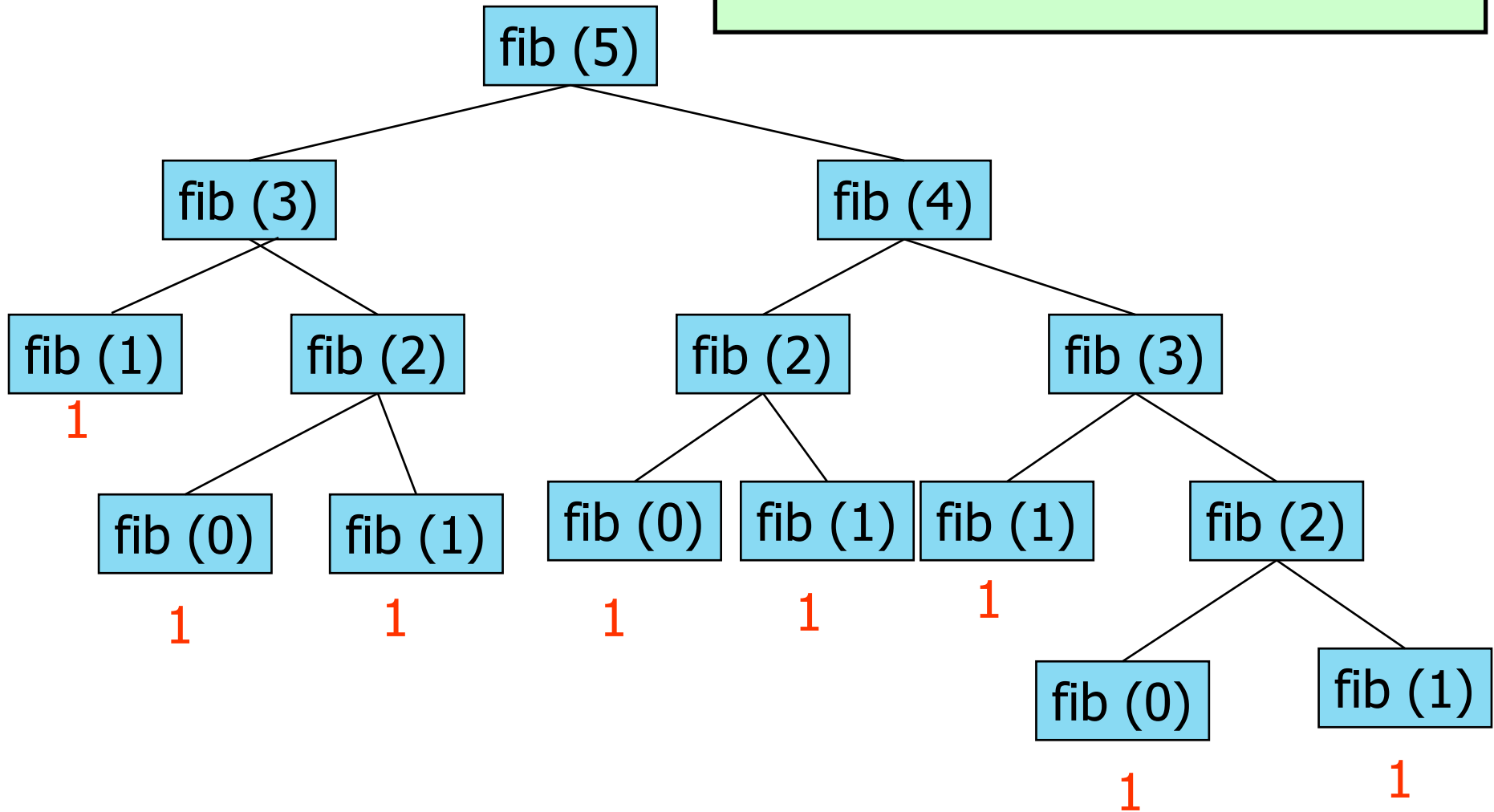
$\text{fib}(n) = 1$ if $n = 0$ or 1 ;
 $= \text{fib}(n - 2) + \text{fib}(n - 1)$
otherwise;



```
int fib (int n) {  
    if (n == 0 || n == 1)  
        return 1;  
    return fib(n-2) + fib(n-1);  
}
```

Fibonacci recurrence:

$\text{fib}(n) = 1$ if $n = 0$ or 1 ;
 $= \text{fib}(n - 2) + \text{fib}(n - 1)$
otherwise;



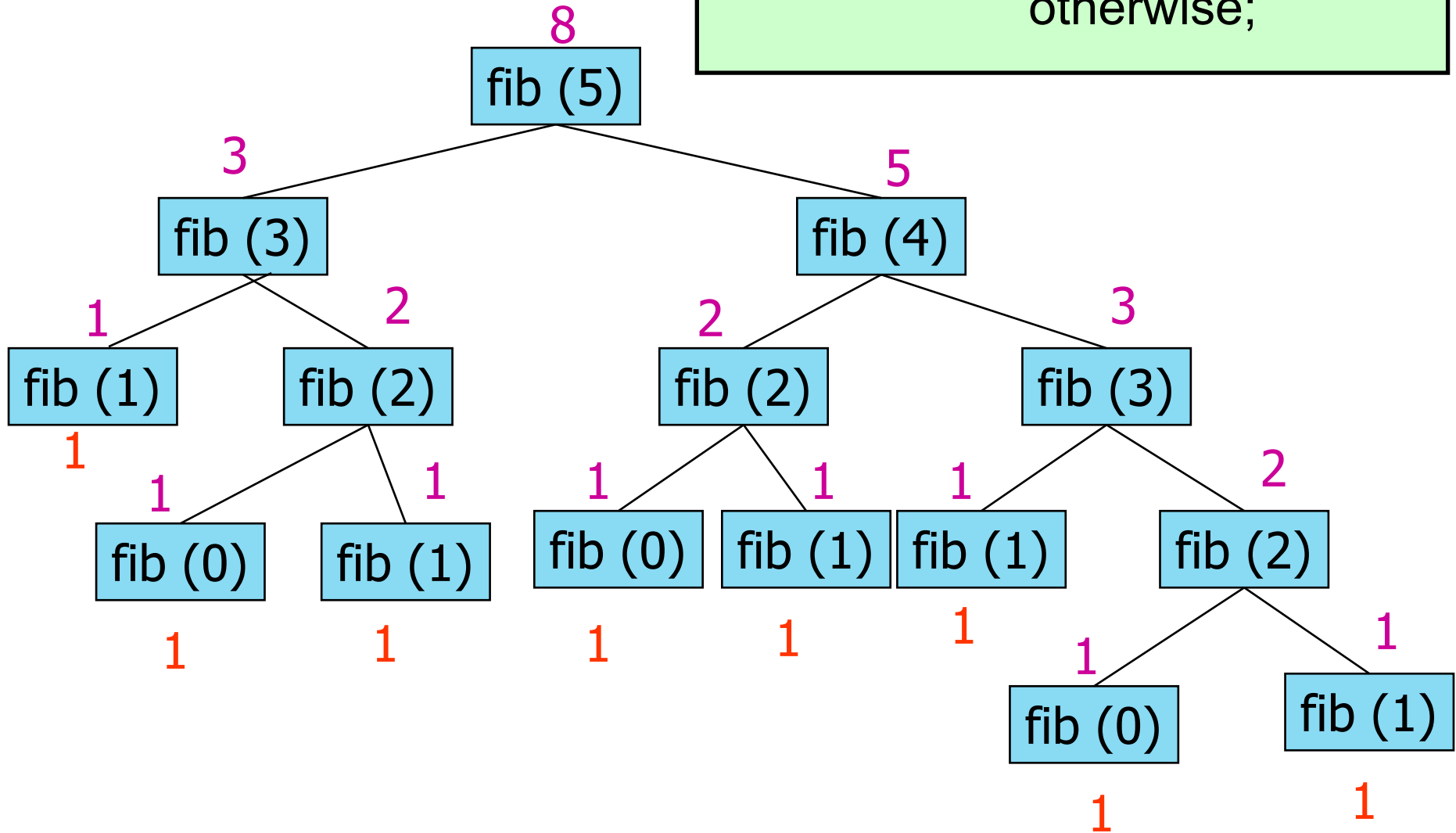
```

int fib (int n)  {
  if (n==0 || n==1)
    return 1;
  return fib(n-2) + fib(n-1);
}

```

Fibonacci recurrence:

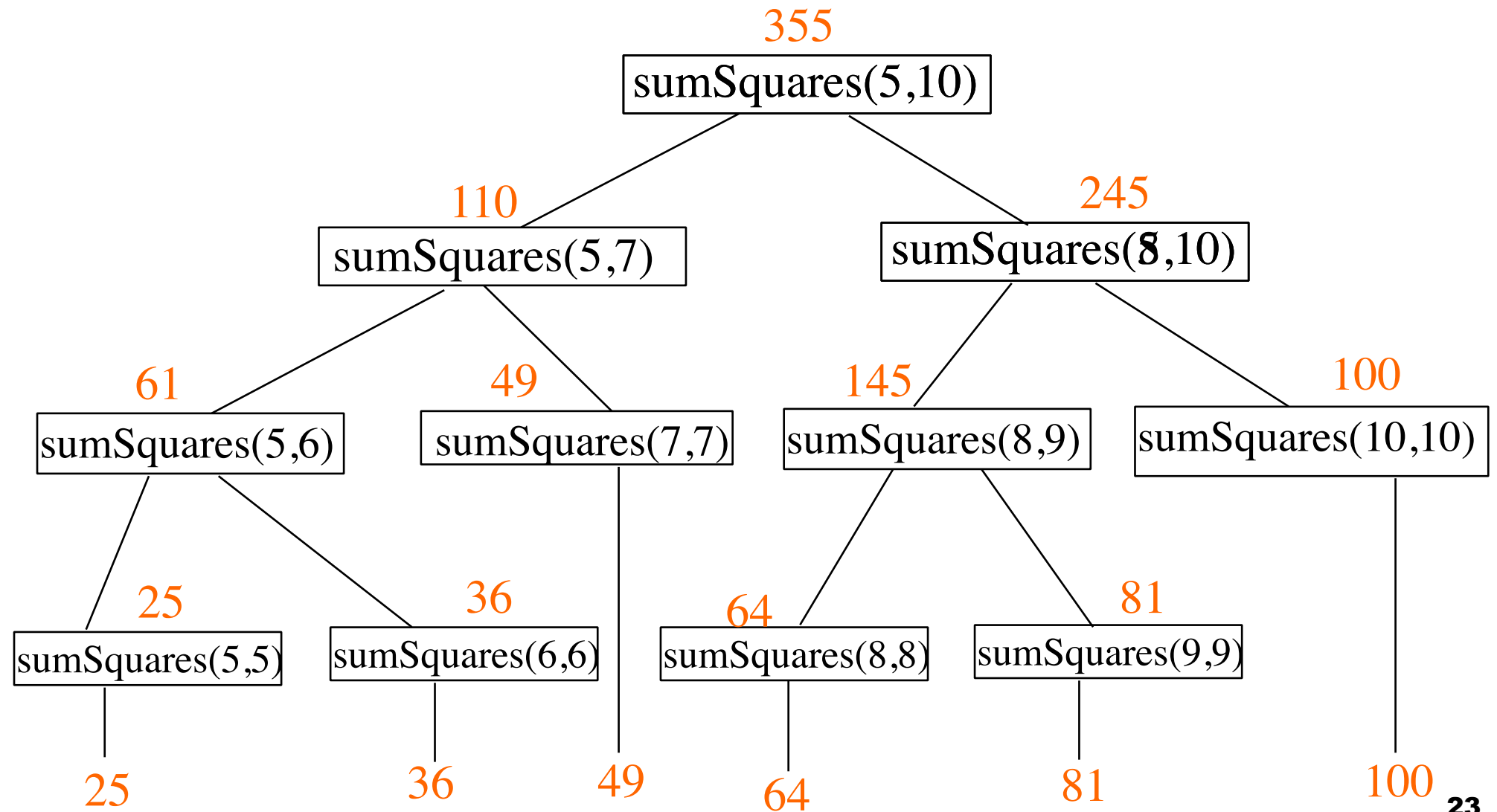
$\text{fib}(n) = 1$ if $n = 0$ or 1 ;
 $= \text{fib}(n - 2) + \text{fib}(n - 1)$
 otherwise;



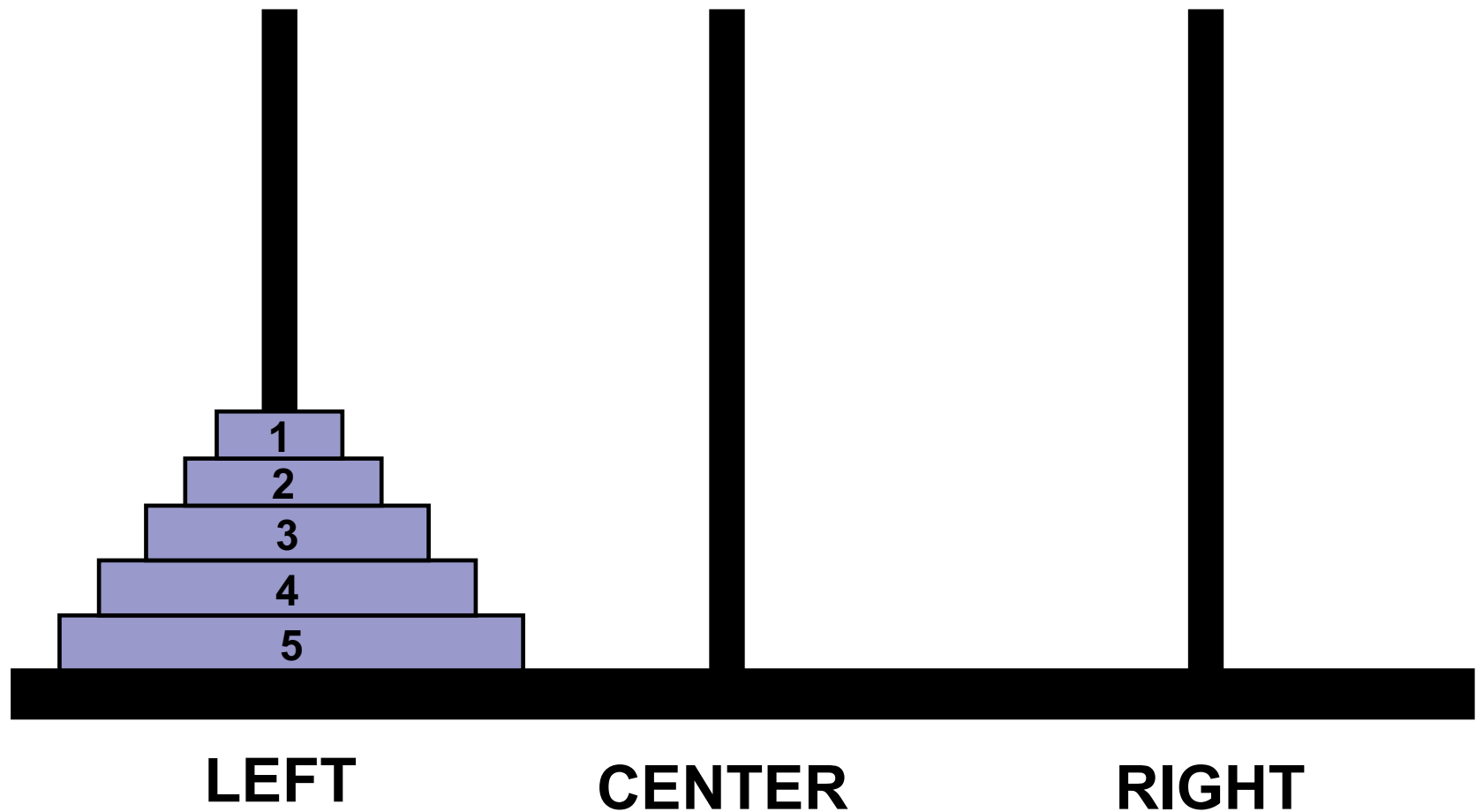
Sum of Squares

```
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



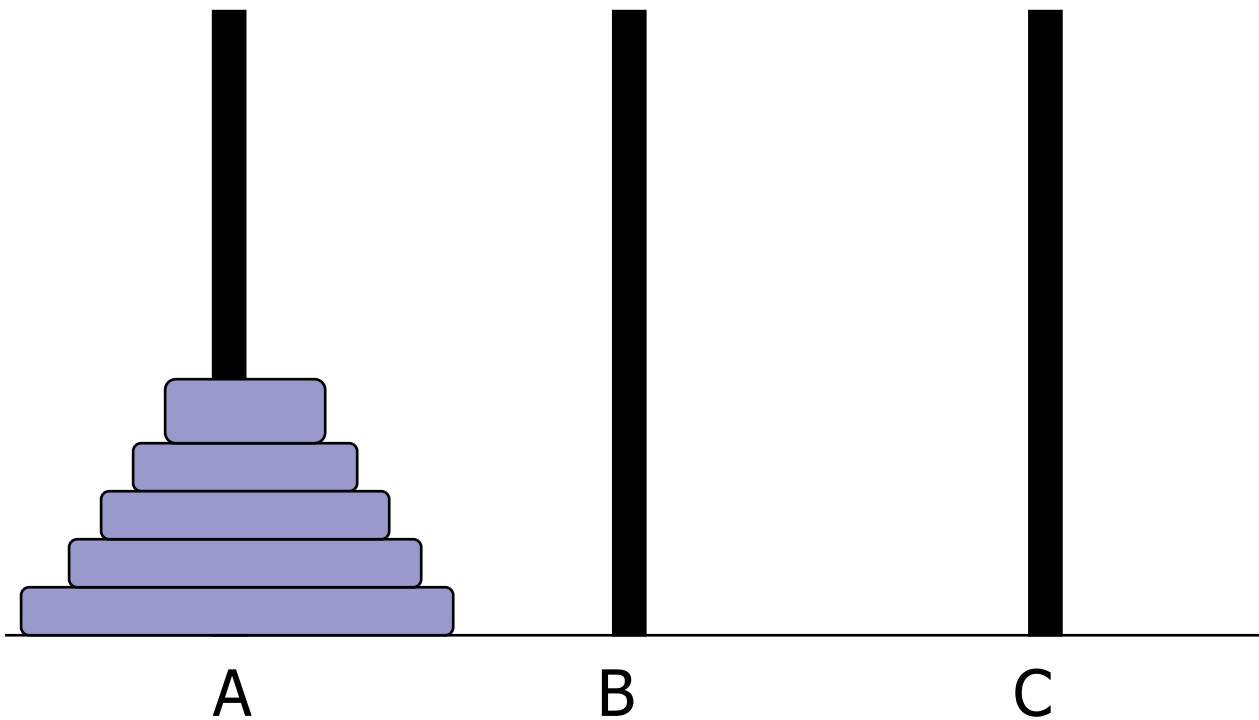
Towers of Hanoi Problem



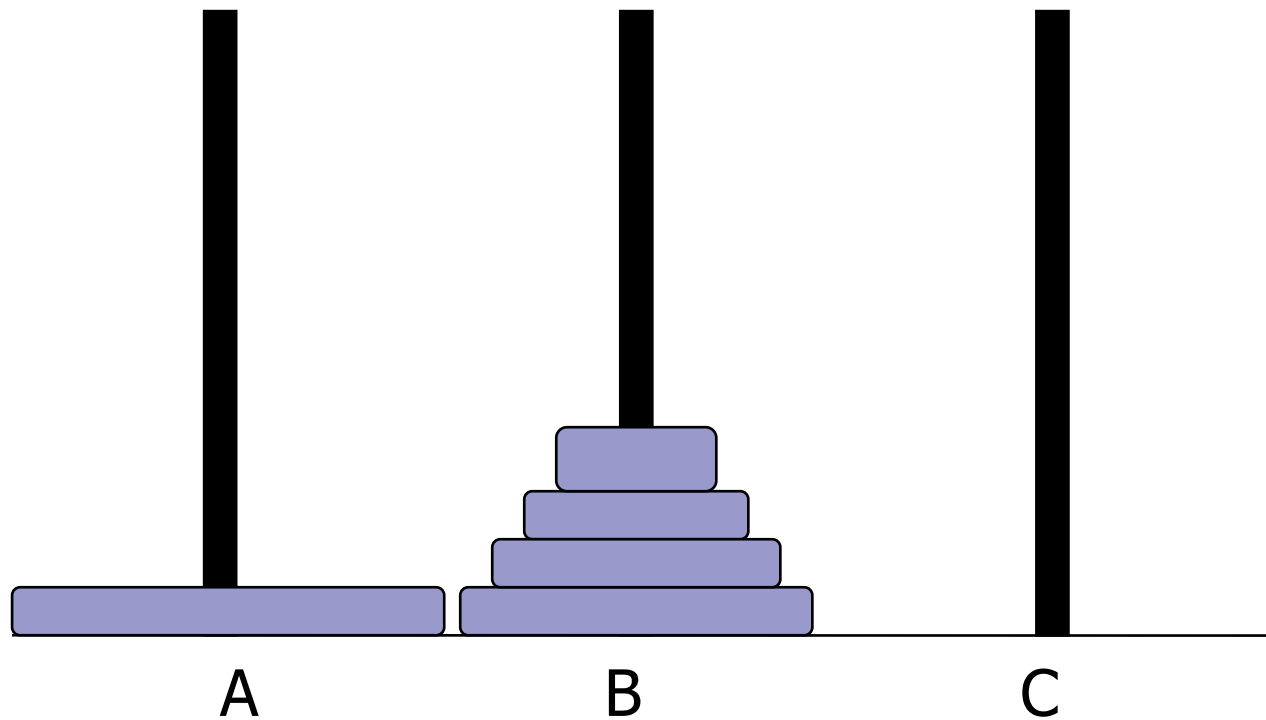
- Initially all the disks are stacked on the LEFT pole
- Required to transfer all the disks to the RIGHT pole
 - Only one disk can be moved at a time.
 - A larger disk cannot be placed on a smaller disk
- CENTER pole is used for temporary storage of disks

- 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

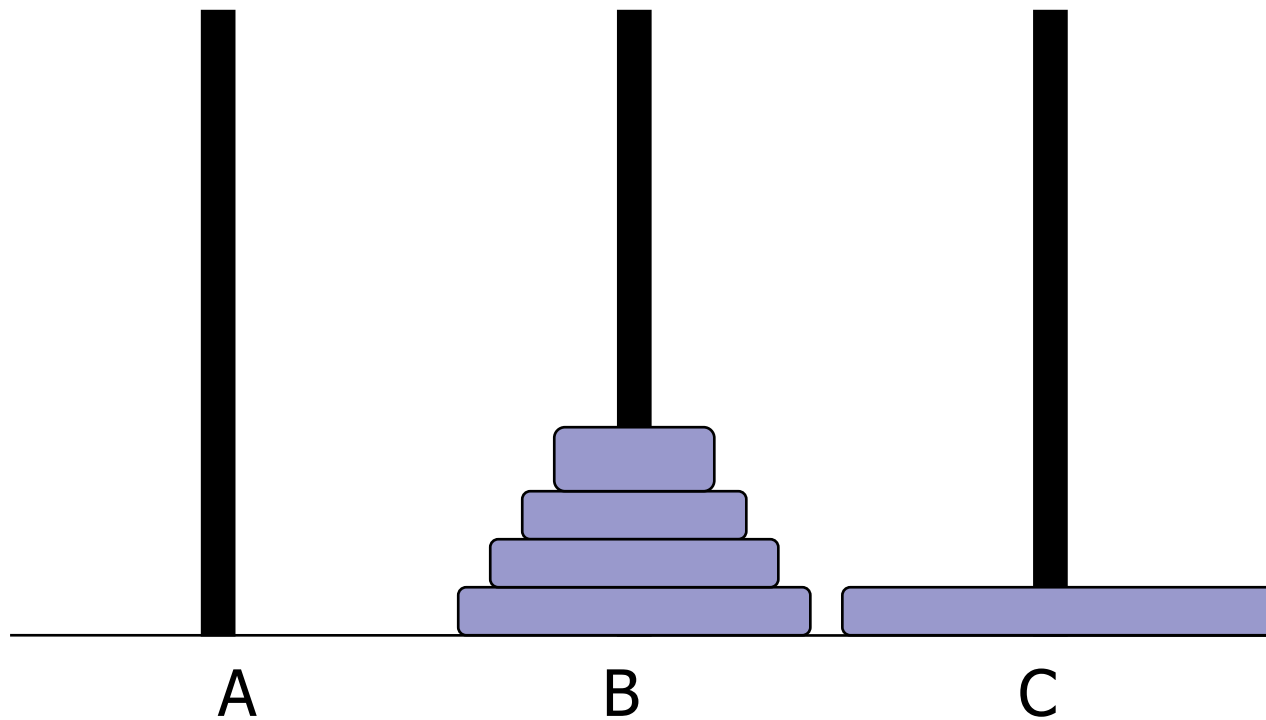
Tower of Hanoi



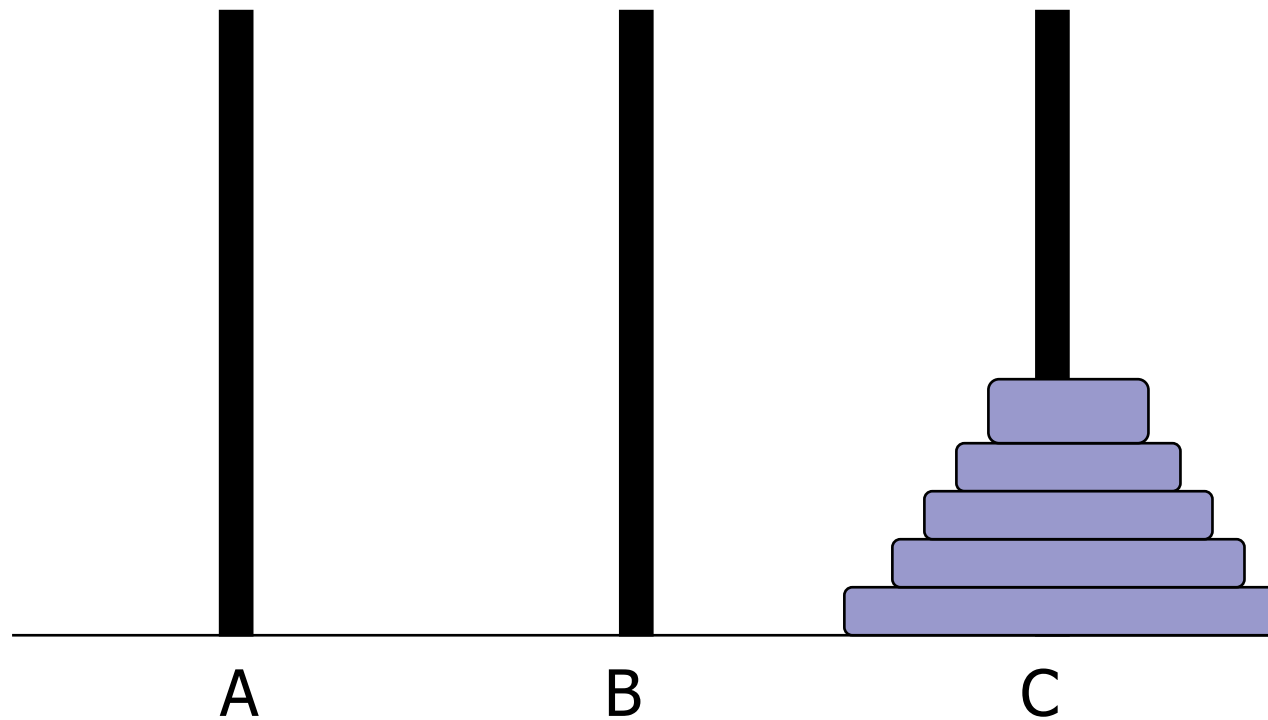
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
    /* Base Condition */
    if (n==1) {
        printf ("Disk 1 : %c → &c \n", from, to) ;
        return ;
    }
    /* Recursive Condition */
    towers (n-1, from, aux, to) ;
    .....
    .....
}
}
```

Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
    /* Base Condition */
    if (n==1) {
        printf ("Disk 1 : %c → %c \n", from, to) ;
        return ;
    }
    /* Recursive Condition */
    towers (n-1, from, aux, to) ;
    printf ("Disk %d : %c → %c\n", n, from, to) ;
    .....
}
```


Towers of Hanoi function

```
void towers (int n, char from, char to, char aux)
{
    /* Base Condition */
    if (n==1) {
        printf ("Disk 1 : %c → %c \n", from, to) ;
        return ;
    }
    /* Recursive Condition */
    towers (n-1, from, aux, to) ;
    printf ("Disk %d : %c → %c\n", n, from, to) ;
    towers (n-1, aux, to, from) ;
}
```

TOH runs

```
void towers(int n, char from, char to, char aux)
{ if (n==1)
  { printf ("Disk 1 : %c -> %c \n", from, to) ;
    return ;
  }
  towers (n-1, from, aux, to) ;
  printf ("Disk %d : %c -> %c\n", n, from, to) ;
  towers (n-1, aux, to, from) ;
}
void main()
{ int n;
  scanf("%d", &n);
  towers(n,'A','C','B');
}
```

Output

```
3
Disk 1 : A -> C
Disk 2 : A -> B
Disk 1 : C -> B
Disk 3 : A -> C
Disk 1 : B -> A
Disk 2 : B -> C
Disk 1 : A -> C
```

More TOH runs

```
void towers(int n, char from, char to, char aux)
{ if (n==1)
  { printf ("Disk 1 : %c -> %c \n", from, to) ;
    return ;
  }
  towers (n-1, from, aux, to) ;
  printf ("Disk %d : %c -> %c\n", n, from, to) ;
  towers (n-1, aux, to, from) ;
}

void main()
{ int n;
  scanf("%d", &n);
  towers(n,'A','C','B');
}
```

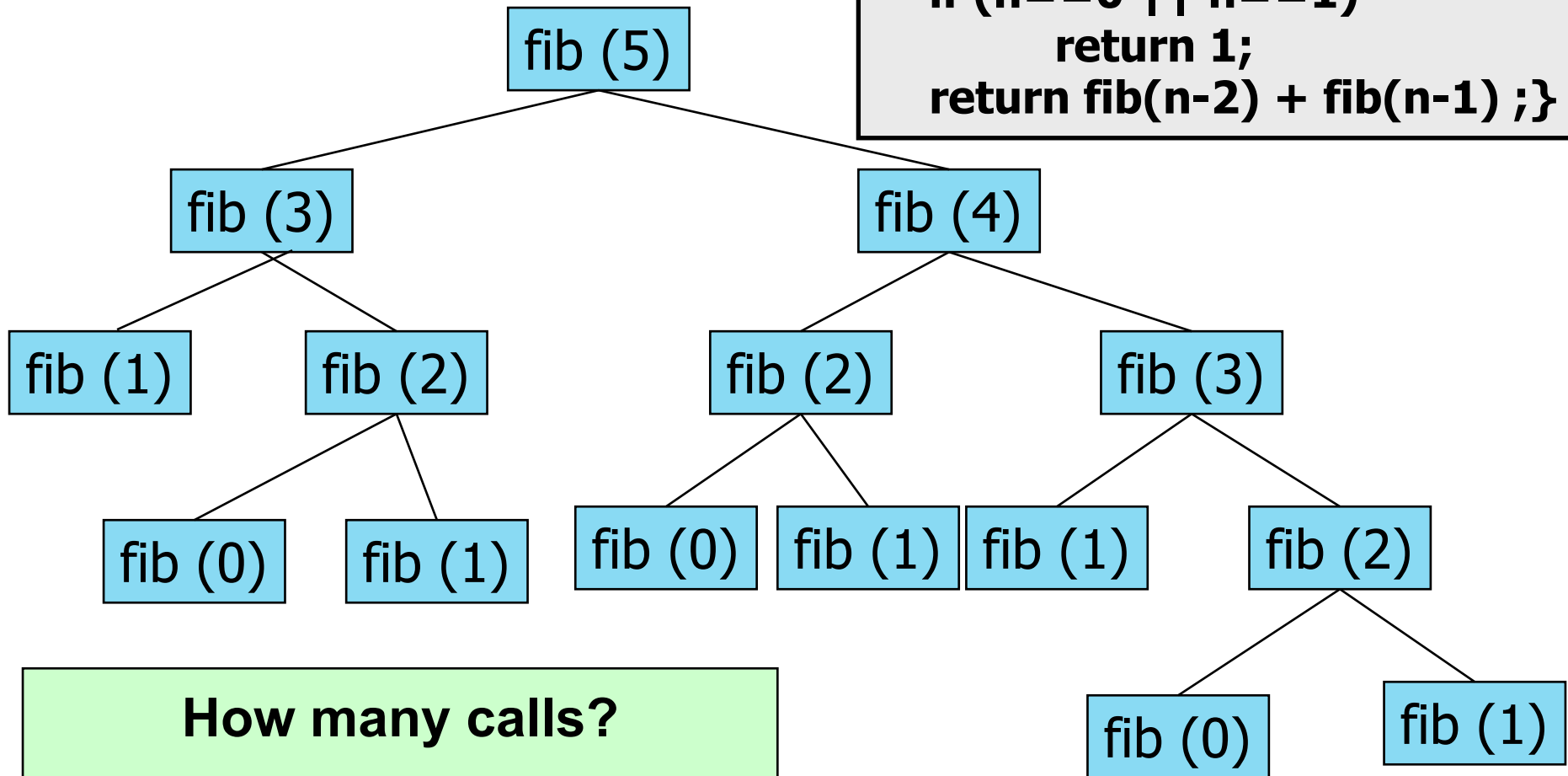
Output

```
4
Disk 1 : A -> B
Disk 2 : A -> C
Disk 1 : B -> C
Disk 3 : A -> B
Disk 1 : C -> A
Disk 2 : C -> B
Disk 1 : A -> B
Disk 4 : A -> C
Disk 1 : B -> C
Disk 2 : B -> A
Disk 1 : C -> A
Disk 3 : B -> C
Disk 1 : A -> B
Disk 2 : A -> C
Disk 1 : B -> C
```

Relook at recursive Fibonacci:

Not efficient !! Same sub-problem solved many times.

```
int fib (int n) {  
    if (n==0 || n==1)  
        return 1;  
    return fib(n-2) + fib(n-1);  
}
```



How many calls?
How many additions?

Iterative Fib

```
int fib( int n)
{ int i=2, res=1, m1=1, m2=1;
  if (n ==0 || n ==1) return res;
  for ( ; i<=n; i++)
  { res = m1 + m2;
    m2 = m1;
    m1 = res;
  }
  return res;
}

void main()
{ int n;
  scanf("%d", &n);
  printf(" Fib(%d) = %d \n", n, fib(n));
}
```

Much Less Computation here!
(How many additions?)

An efficient recursive Fib

```
int Fib ( int, int, int, int);

void main()
{
    int n;
    scanf("%d", &n);
    if (n == 0 || n ==1)
        printf("F(%d) = %d \n", n, 1);
    else
        printf("F(%d) = %d \n", n, Fib(1,1,n,2));
}
```

```
int Fib(int m1, int m2, int n, int i)
{
    int res;
    if (n == i)
        res = m1+ m2;
    else
        res = Fib(m1+m2, m1, n, i+1);
    return res;
}
```

**Much Less Computation here!
(How many calls/additions?)**

Run

```
int Fib ( int, int, int, int);
void main()
{ int n;
  scanf("%d", &n);
  if (n == 0 || n ==1) printf("F(%d) = %d \n", n, 1);
  else printf("F(%d) = %d \n", n, Fib(1,1,n,2));
}
int Fib(int m1, int m2, int n, int i)
{ int res;
  printf("F: m1=%d, m2=%d, n=%d, i=%d\n",
        m1,m2,n,i);
  if (n == i)
    res = m1+ m2;
  else
    res = Fib(m1+m2, m1, n, i+1);
  return res;
}
```

Output

```
$ ./a.out
```

```
3
```

```
F: m1=1, m2=1, n=3, i=2
```

```
F: m1=2, m2=1, n=3, i=3
```

```
F(3) = 3
```

```
$ ./a.out
```

```
5
```

```
F: m1=1, m2=1, n=5, i=2
```

```
F: m1=2, m2=1, n=5, i=3
```

```
F: m1=3, m2=2, n=5, i=4
```

```
F: m1=5, m2=3, n=5, i=5
```

```
F(5) = 8
```

Static Variables

```
int Fib (int, int);

void main()
{
    int n;
    scanf("%d", &n);
    if (n == 0 || n ==1)
        printf("F(%d) = %d \n", n, 1);
    else
        printf("F(%d) = %d \n", n,
Fib(n,2));
}
```

```
int Fib(int n, int i)
{
    static int m1, m2;
    int res, temp;
    if (i==2) {m1 =1; m2=1;}
    if (n == i) res = m1+ m2;
    else
        { temp = m1;
          m1 = m1+m2;
          m2 = temp;
          res = Fib(n, i+1);
        }
    return res;
}
```

Static variables remain in existence rather than coming and going each time a function is activated

Static Variables: See the addresses!


```
int Fib(int n, int i)
{
    static int m1, m2;
    int res, temp;
    if (i==2) {m1 =1; m2=1;}
    printf("F: m1=%d, m2=%d, n=%d,
           i=%d\n", m1,m2,n,i);
    printf("F: &m1=%u, &m2=%u\n",
           &m1,&m2);
    printf("F: &res=%u, &temp=%u\n",
           &res,&temp);
    if (n == i) res = m1+ m2;
    else { temp = m1; m1 = m1+m2;
          m2 = temp;
          res = Fib(n, i+1);  }
    return res;
}
```

Output

```
5
F: m1=1, m2=1, n=5, i=2
F: &m1=134518656, &m2=134518660
F: &res=3221224516, &temp=3221224512
F: m1=2, m2=1, n=5, i=3
F: &m1=134518656, &m2=134518660
F: &res=3221224468, &temp=3221224464
F: m1=3, m2=2, n=5, i=4
F: &m1=134518656, &m2=134518660
F: &res=3221224420, &temp=3221224416
F: m1=5, m2=3, n=5, i=5
F: &m1=134518656, &m2=134518660
F: &res=3221224372, &temp=3221224368
F(5) = 8
```

Recursion vs. 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).

- 
- 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

How are function calls implemented?

- The following applies in general, with minor variations that are implementation dependent
 - The system maintains a stack in memory
 - Stack is a last-in first-out structure
 - Two operations on stack, push and pop
 - Whenever there is a function call, the activation record gets pushed into the stack
 - Activation record consists of the return address in the calling program, the return value from the function, and the local variables inside the function

```
void main()
{
    .....
    x = gcd (a, b);
    .....
}
```

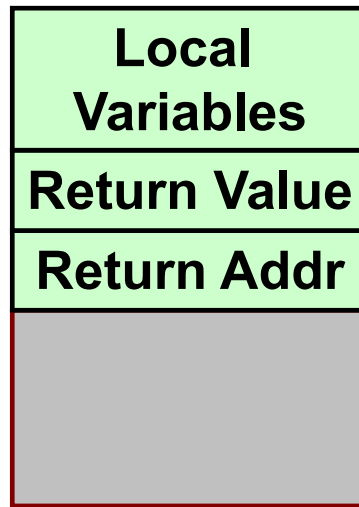
```
int gcd (int x, int y)
{
    .....
    .....
    return (result);
}
```

STACK

Activation record



Before call



After call

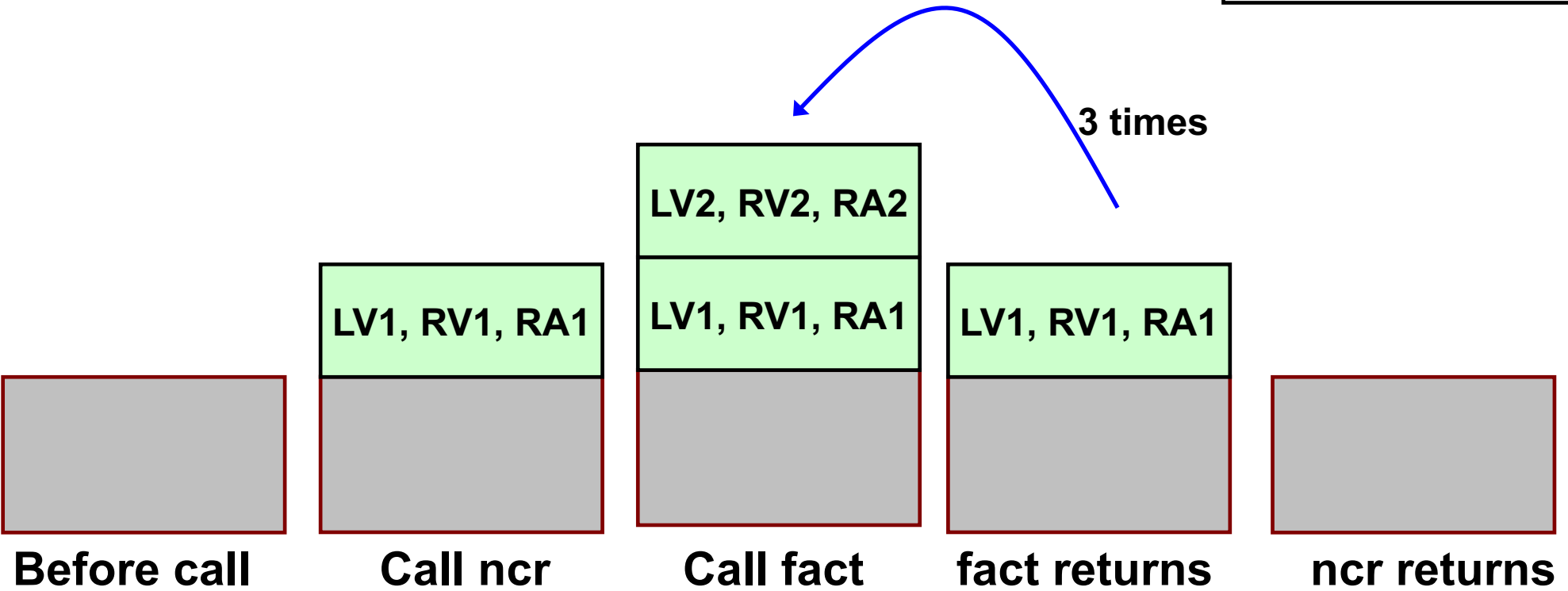


After return

```
void main()
{
    .....
    x = ncr (a, b);
    .....
}
```

```
int ncr (int n, int r)
{
    return (fact(n)/
           fact(r)/fact(n-r));
}
```

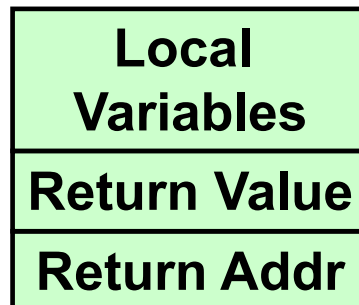
```
int fact (int n)
{
    .....
    return (result);
}
```



What happens for recursive calls?

- 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:

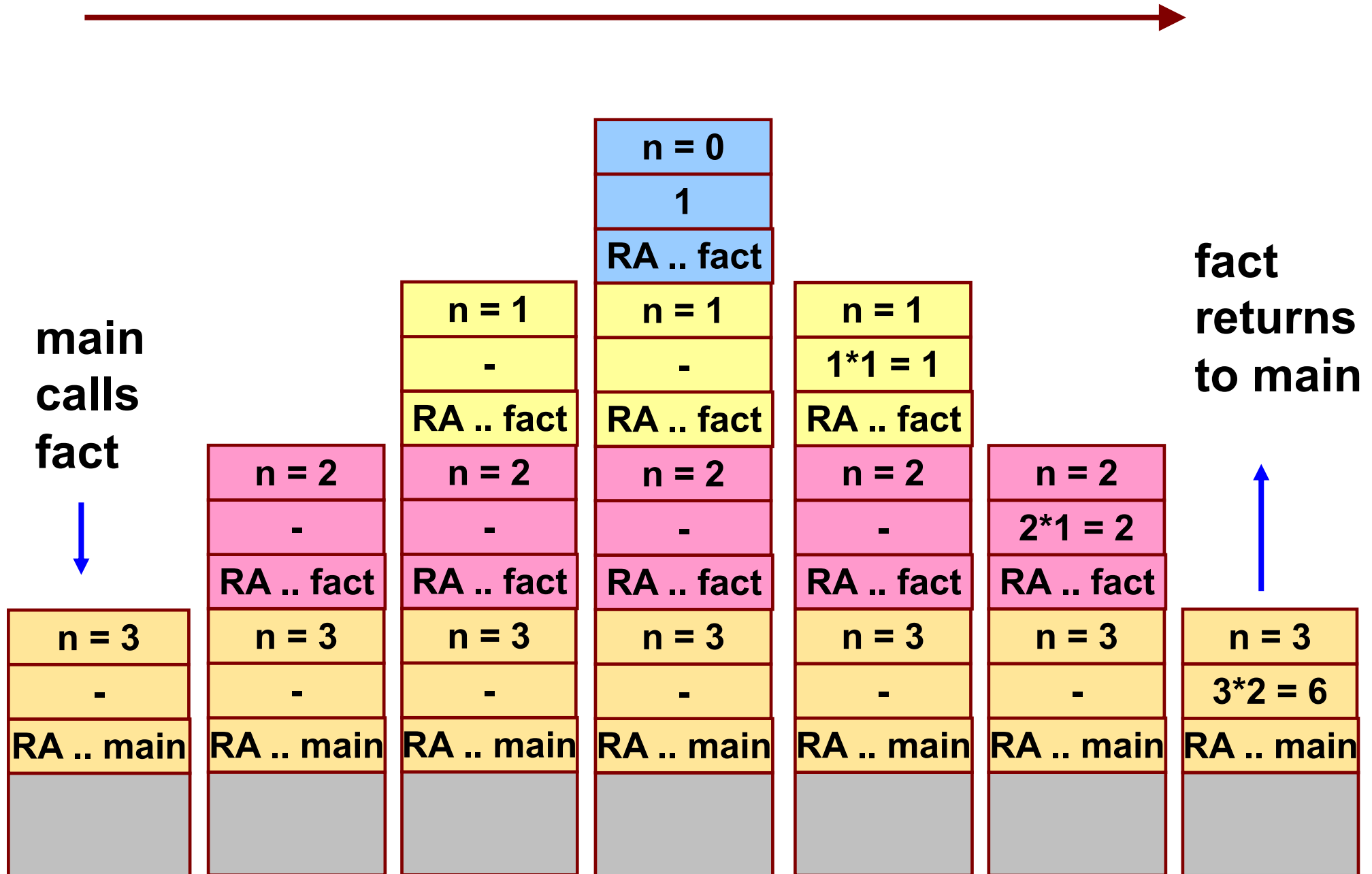


Example:: main() calls fact(3)

```
void 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

```
int f (int n)
{
    int a, b;
    if (n < 2) return (n);
    else {
        a = f(n-1);
        b = f(n-2);
        return (a+b);
    }
}
```

X



Y



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

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

main

