Recursion

1

CS101

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

  fact(n) = n * 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

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

- Examples:

  - Factorial:

    $fact(0) = 1$

    $fact(n) = n * fact(n-1)$, if $n > 0$

  - 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 $n > 1$

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));                  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);
```

6

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));                          24

    if   (3 = = 1) return (1);
    else return  (3 * fact(2));                      6

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

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

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

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]
 }
```

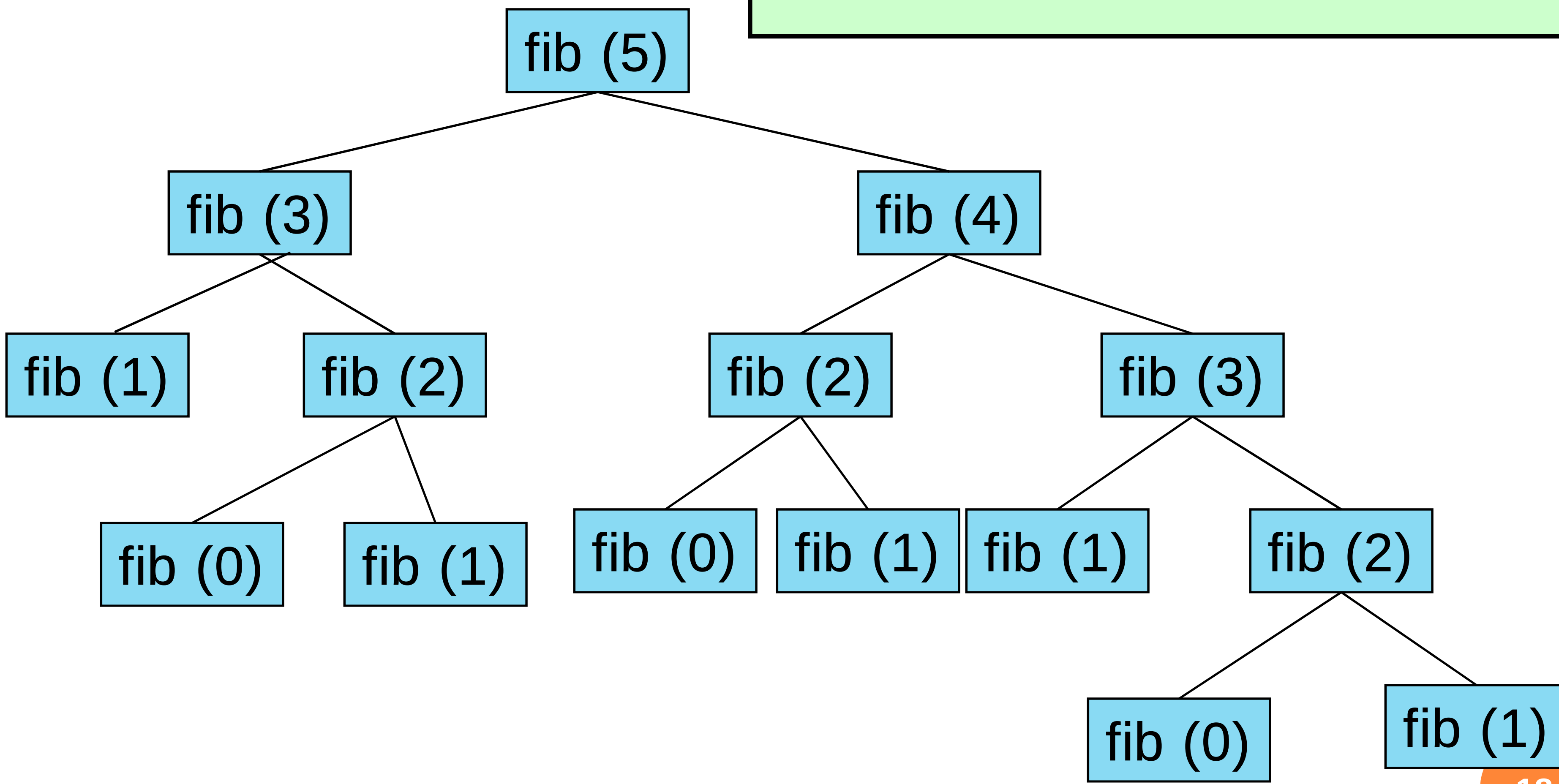
```
int fib (int n)     {
   if (n == 0 || n == 1)
          return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

otherwise;



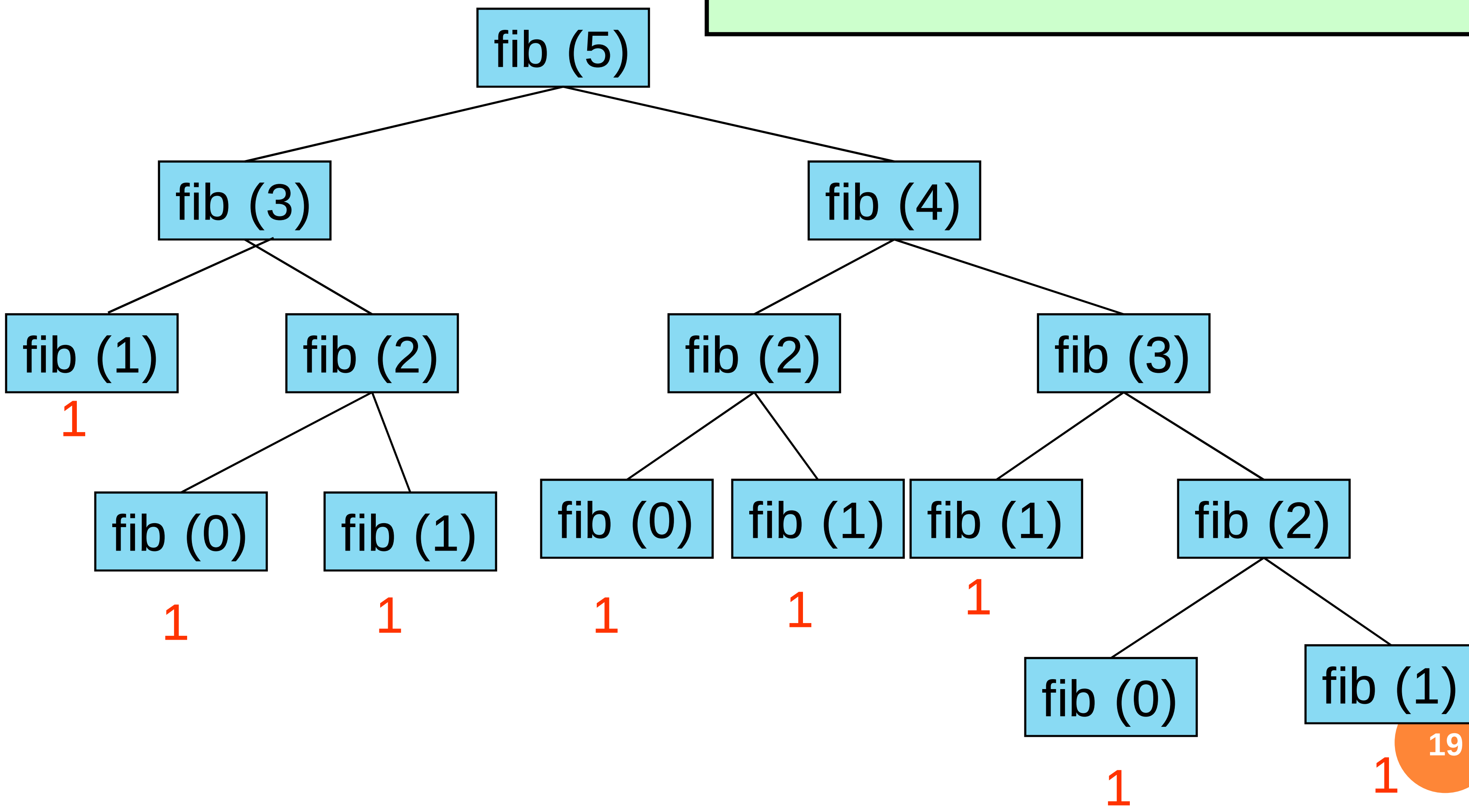
```c
int fib (int n)    {
   if (n == 0 || n == 1)
         return 1;
   return fib(n-2) + fib(n-1) ;
}
```

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1)

otherwise;



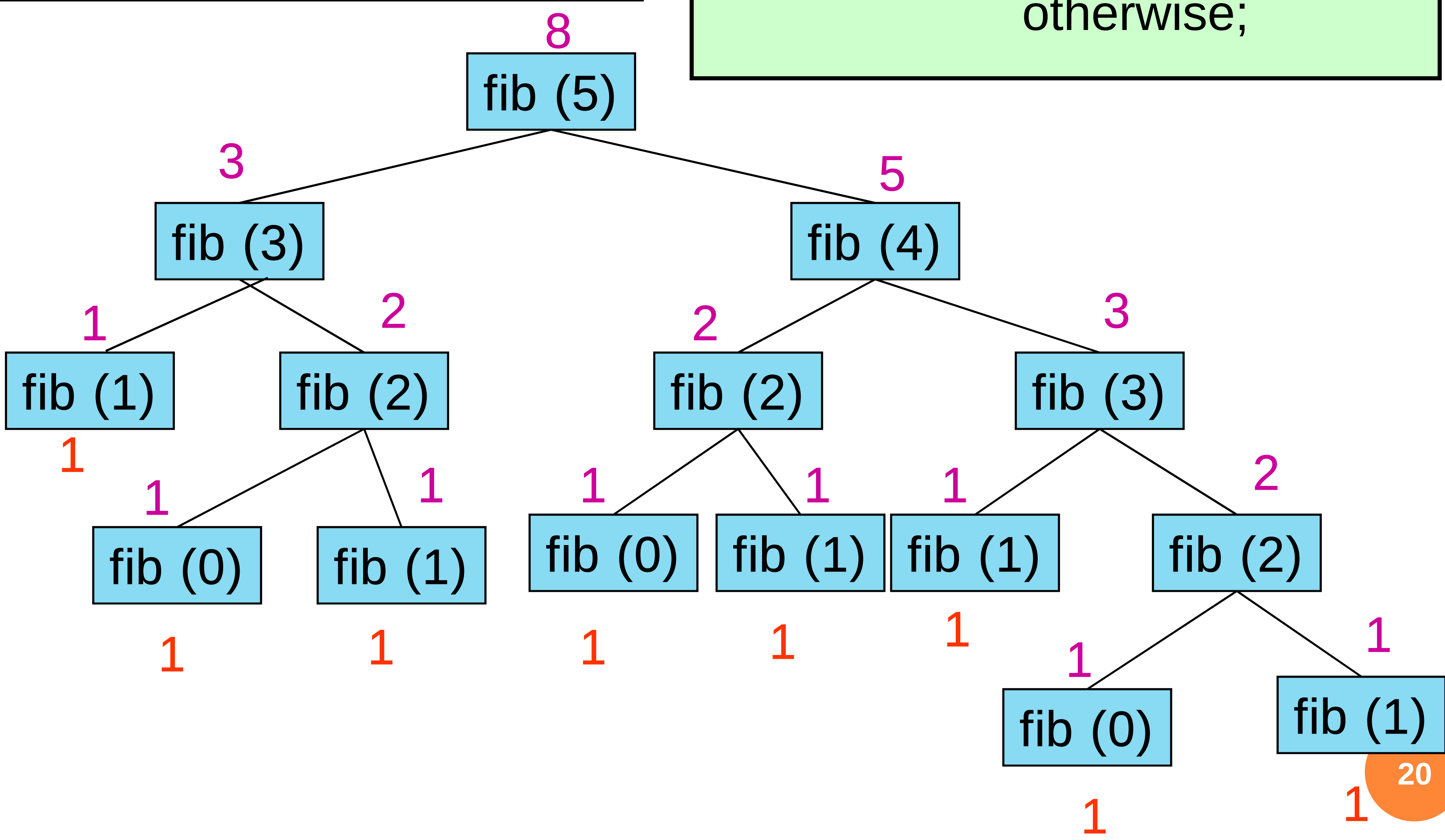
fib.c

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

Fibonacci recurrence:

fib(n) = 1 if n = 0 or 1;

= fib(n – 2) + fib(n – 1) otherwise;



EXAMPLE: SUM OF SQUARES

- Write a recursive function

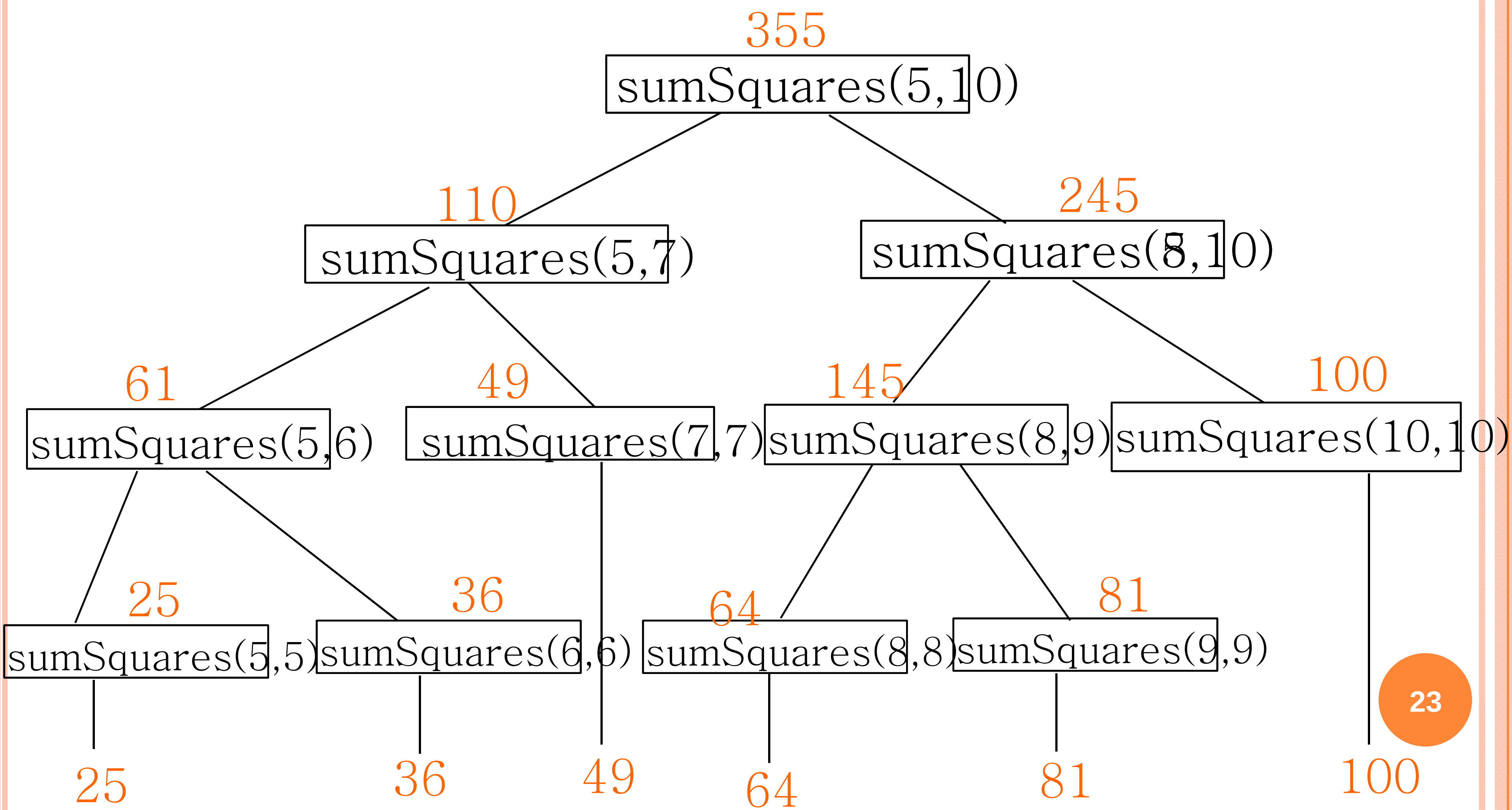
  int sumSquares(int m, int n) where n >= m

  - to compute $m^2 + (m+1)^2 + \ldots + n^2$
- So a call to sumSquares(5,10) will eventually return the result of
  - $5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2$

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);
     }
}
```

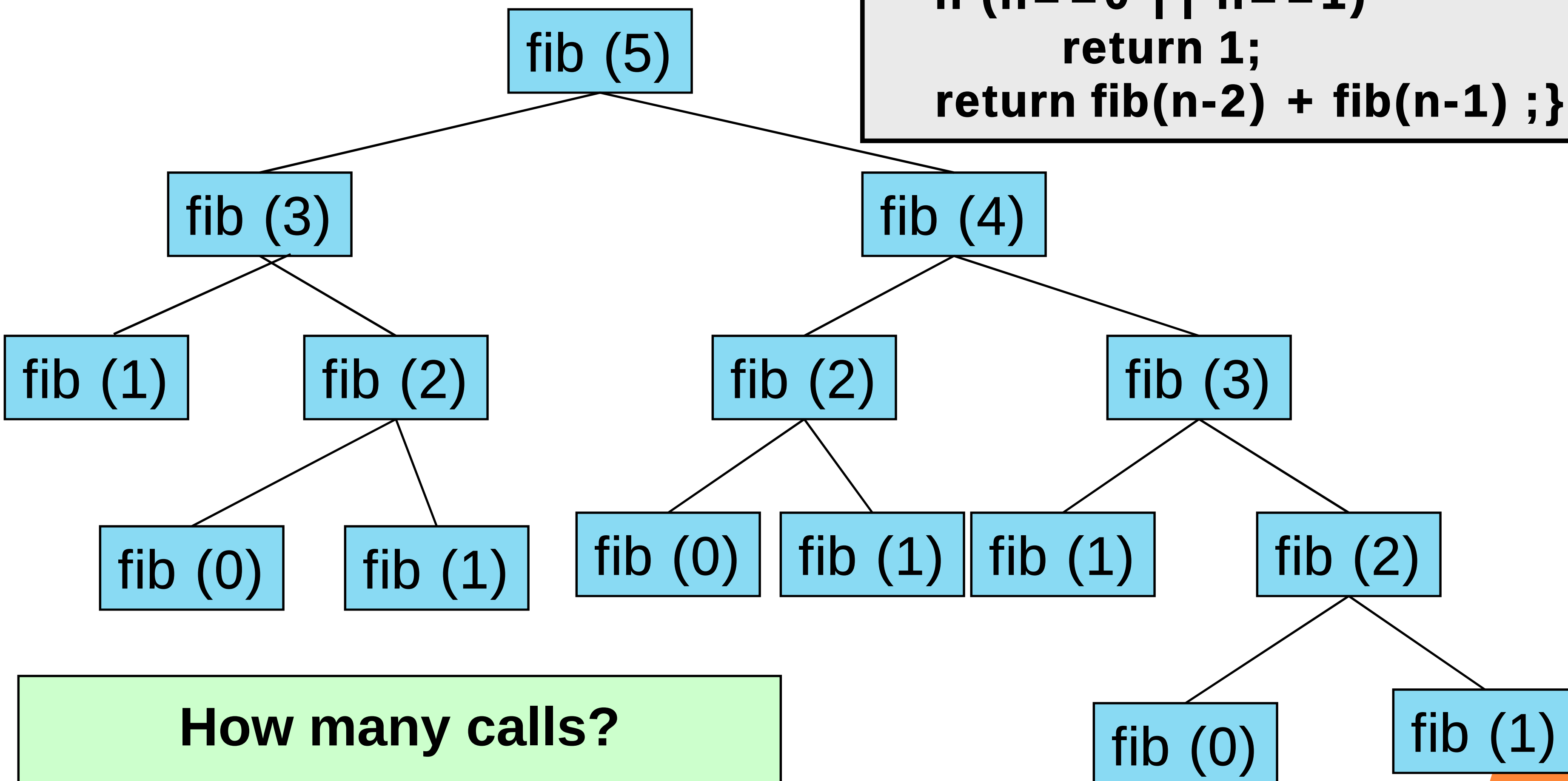
Annotated Call Tree



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).

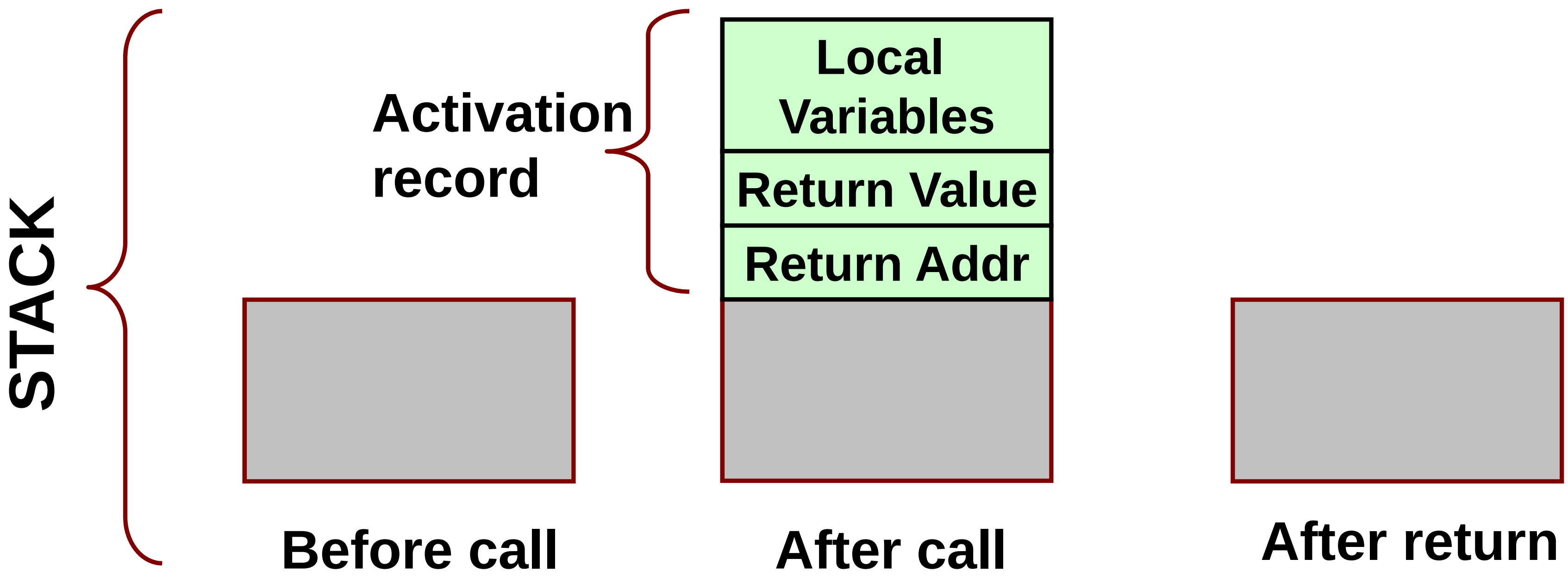
- Every recursive program can also be written without recursion
- Recursion is used for programming convenience, not for performance enhancement
- Sometimes, if the function being computed has a nice recurrence form, then a recursive code may be more readable

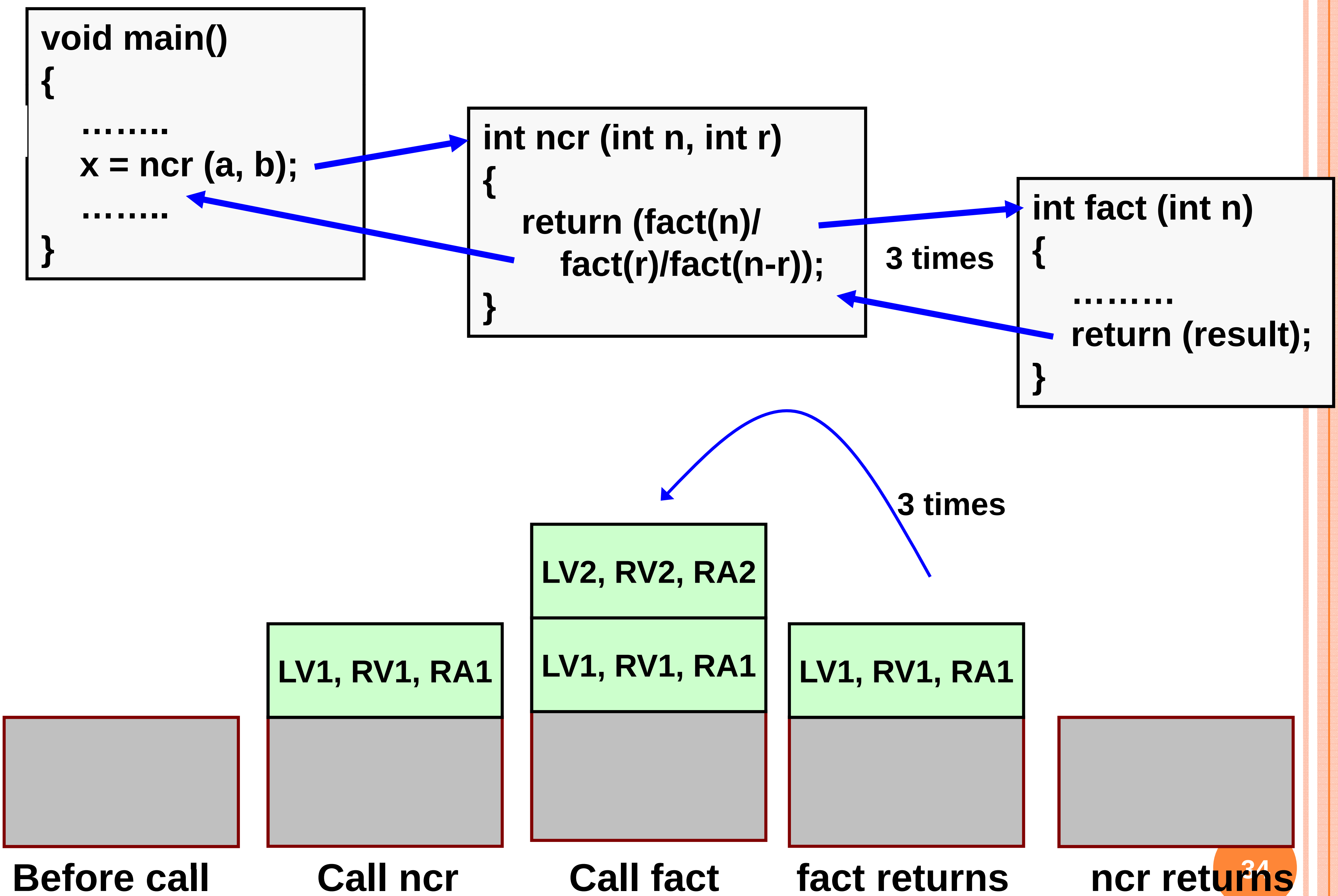
How are function calls implemented?

- 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);
}
```




void main()
{
....
x = ncr (a, b);
....
}
int ncr (int n, int r)
{
return (fact(n)/
fact(r)/fact(n-r));
}
3 times
int fact (int n)
{
....
return (result);
}
3 times
LV2, RV2, RA2
LV1, RV1, RA1
LV1, RV1, RA1
LV1, RV1, RA1
Before call
Call ncr
Call fact
fact returns
ncr returns

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:

Local Variables
Return Value
Return Addr

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