CS11001/CS11002 Programming and Data Structures (PDS) (Theory: 3-0-0)

Teacher: Sourangshu Bhattacharya
sourangshu@gmail.com
http://cse.iitkgp.ac.in/~sourangshu/

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

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)

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 – in terms of problems of lower size.
  - The problem statement must include a stopping condition

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

Recursion

- 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

Facts on fact

- `5! = 5 * 4 * 3 * 2 * 1`
- Notice that
  - `5! = 5 * 4!`
  - `4! = 4 * 3! ...`
- Can compute factorials recursively
- Solve base case (`1! = 0! = 1`) then plug in
  - `2! = 2 * 1! = 2 * 1 = 2;`
  - `3! = 3 * 2! = 3 * 2 = 6;`

Example 1 :: Factorial

```
#include <stdio.h>

int fact(int n)
{
   if (n == 0)
      return 1;
   else
      return (n * fact(n-1));
}
void main()
{
    int i=6;
    printf (“Factorial of 6 is: %d \n”, 
fact(i));
}
```

Mechanism of Execution

- When a recursive program is executed, the recursive function calls are not executed immediately.
  - They are kept aside (on a stack) until the stopping condition is encountered.
  - The function calls are then executed in reverse order.

Advantage of Recursion :: Calculating fact(5)

– First, the function calls will be processed:

fact(5) = 5 * fact(4)

fact(4) = 4 * fact(3)

fact(3) = 3 * fact(2)

fact(2) = 2 * fact(1)

fact(1) = 1 * fact(0)

– The actual values return in the reverse order:

fact(0) = 1

fact(1) = 1 * 1 = 1

fact(2) = 2 * 1 = 2

fact(3) = 3 * 2 = 6

fact(4) = 4 * 6 = 24

Fact(5) = 5 * 24 = 120

Example 2 :: Fibonacci series

```
#include <stdio.h>

int fib(int n)
{
   if (n < 2)
         return n;
   else
         return (fib(n-1) + fib(n-2));
}
void main()
{
    int i=4;
    printf (“%d \n”, fib(i));
}
```

Execution of Fibonacci number

- Fibonacci number fib(n) can be defined as:

$$fib(0) = 0$$

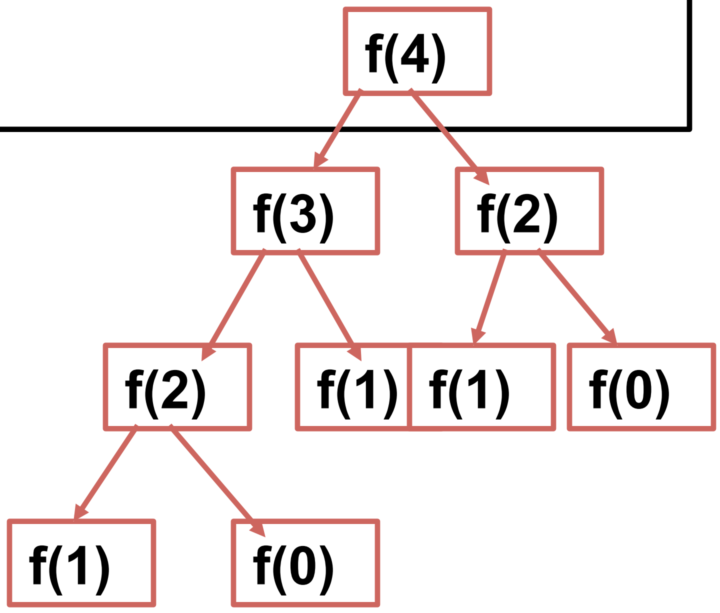
$$fib(1) = 1$$

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

  - The successive Fibonacci numbers are:

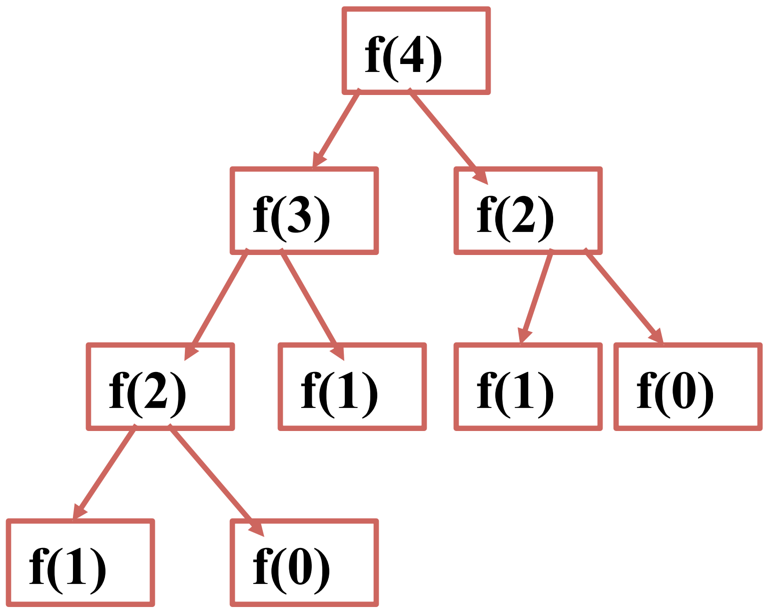
    0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
int  fib(int n)
{
    if  (n  < 2)
      return (n);
    else
      return (fib(n-1) + fib(n-2));
}
```



Inefficiency of Recursion

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

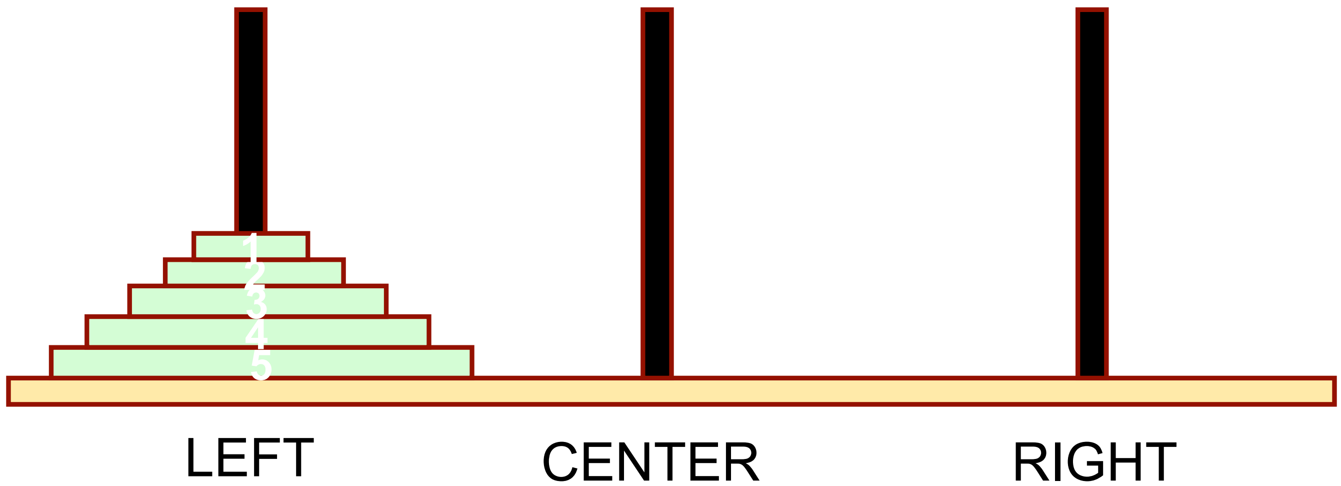


- Same thing is computed several times.

Performance Tip

- Avoid Fibonacci-style recursive programs which result in an exponential “explosion” of calls.

Example 3: 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 can be moved at a time.
    - A larger disk cannot be placed on a smaller disk.

Recursion is implicit

- 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.

Recursive C code: Towers of Hanoi

```
#include  <stdio.h>

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

int main()
{
   int  n;  /* Number of disks */
   scanf (“%d”, &n);
   transfer (n, ‘L’, ‘R’, ‘C’);
   return 0;
}

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;
}
```

Towers of Hanoi: Example Run

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

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

5
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
Move disk 4 from L to C
Move disk 1 from R to C
Move disk 2 from R to L
Move disk 1 from C to L
Move disk 3 from R to C
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 5 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
Move disk 3 from C to L
Move disk 1 from R to C
Move disk 2 from R to L
Move disk 1 from C to L
Move disk 4 from C to R
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

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)

Performance Tip

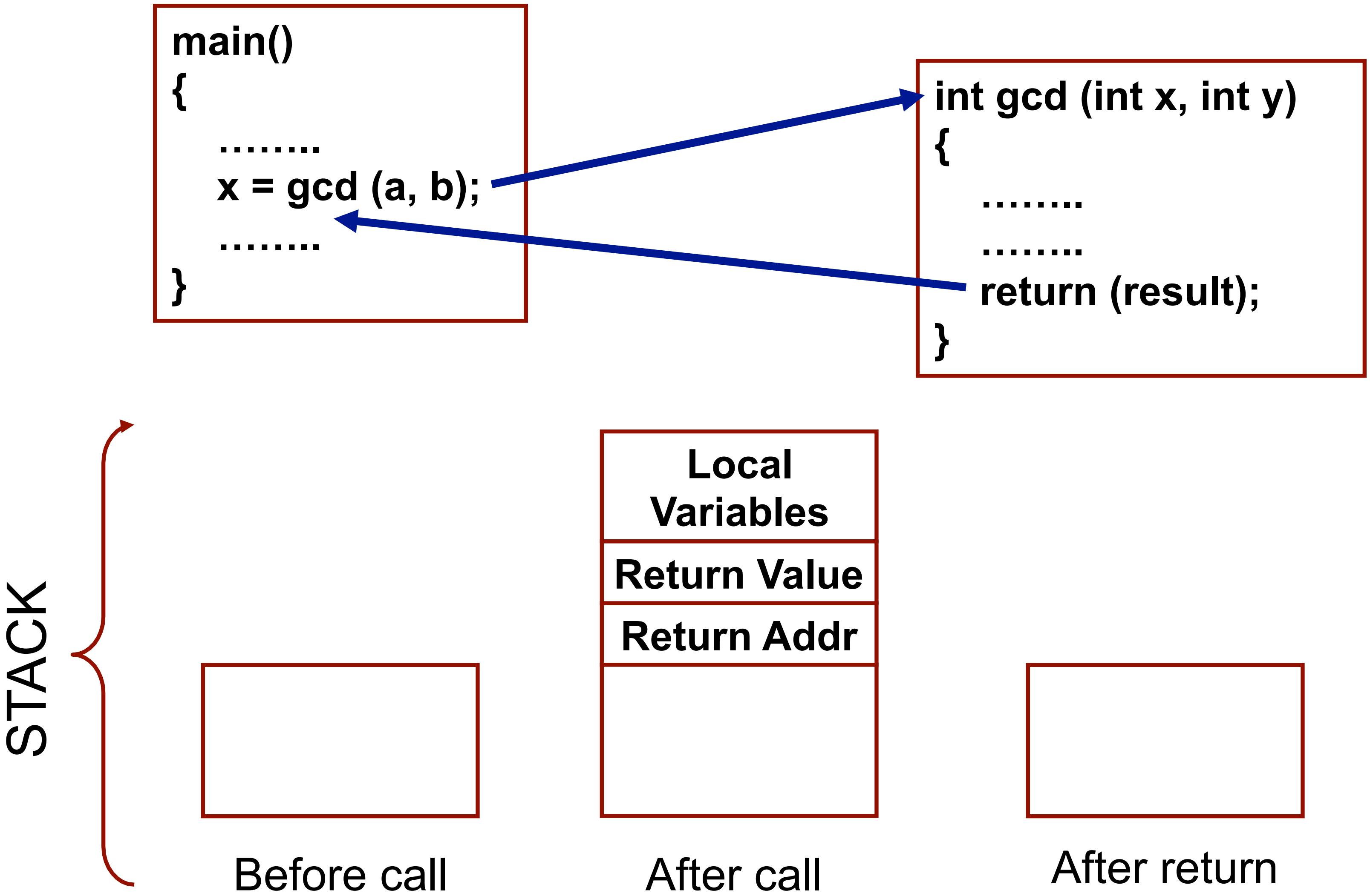
- Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

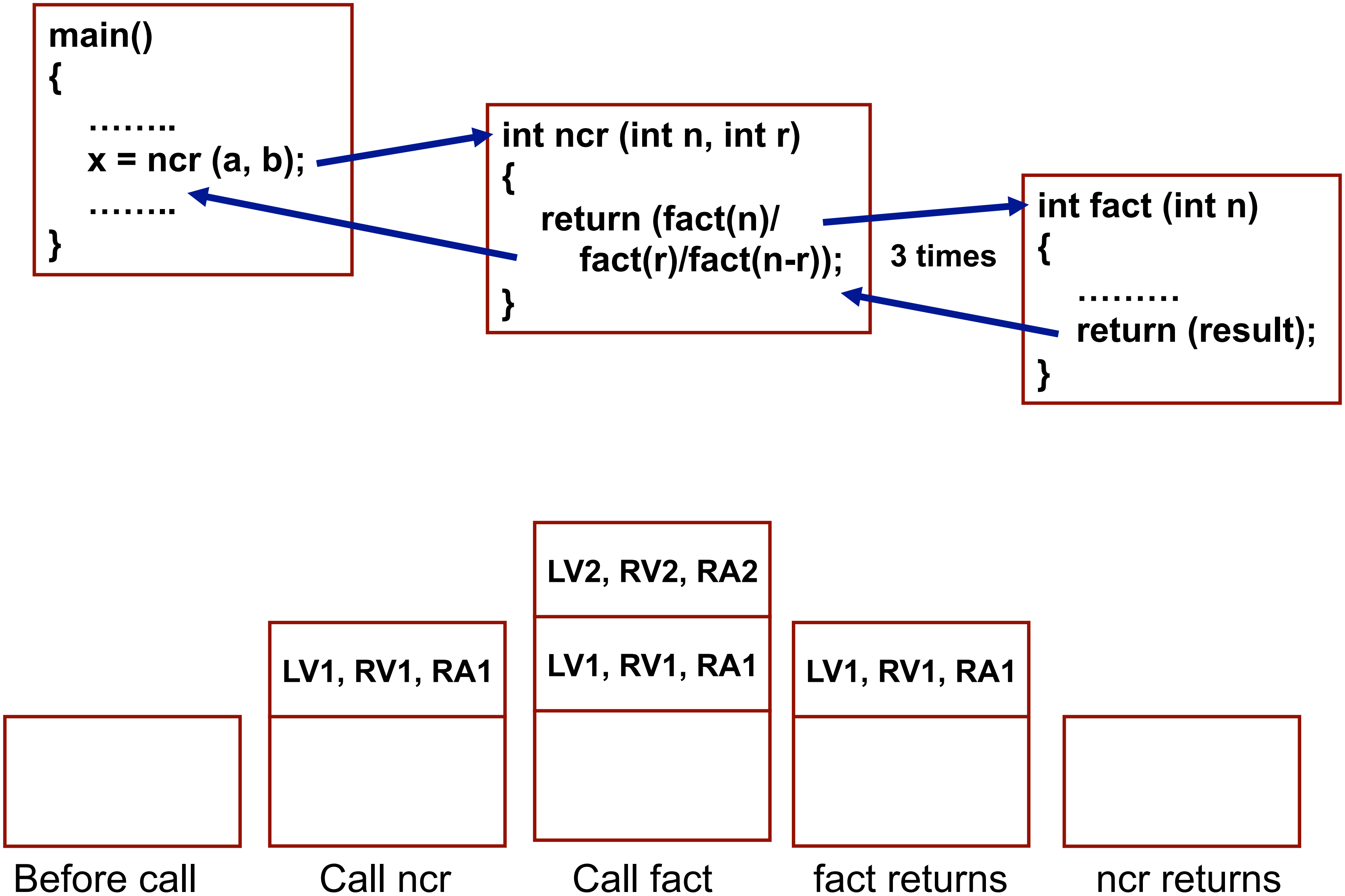
How are function calls implemented?

- In general, during program execution
  - 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.
    - At the end of function call, the corresponding *activation record* gets *popped* out of the stack.

At the system



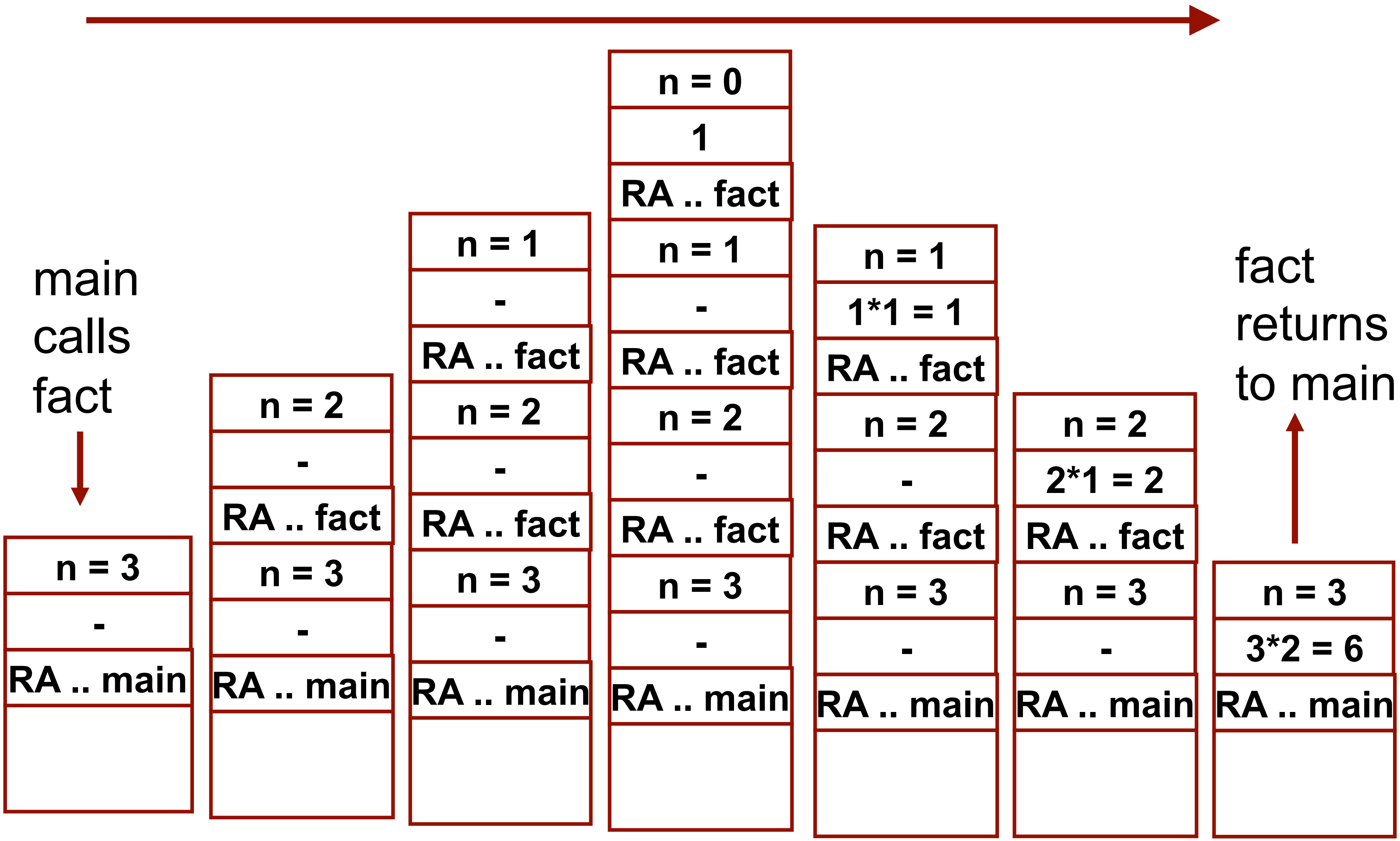

main()
{
....
x = ncr (a, b);
....
}
int ncr (int n, int r)
{
return (fact(n)/
fact(r)/fact(n-r));
}
3 times
int fact (int n)
{
....
return (result);
}
LV2, RV2, RA2
LV1, RV1, RA1
LV1, RV1, RA1
LV1, RV1, RA1
Before call
Call ncr
Call fact
fact returns
ncr returns

Example:: main() calls fact(3)

```
void main()
{
    int  n;
    n = 4;
    printf ("%d \n", fact(n) );
}
```

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

TRACE OF THE STACK DURING EXECUTION



Homework

Trace of Execution for Fibonacci Series