Recursion

Comp Sci 1575 Data Structures

Missouri S&T | Computer Science

Outline

Definitions

"To understand recursion, you must understand recursion."

Familiar examples of recursive definitions

- Natural numbers are either:
  $n + 1$, where n is a natural number, or
  1
- Exponentiation:
  $b^n = b * b^{n-1}$, or
  $b^0 = 1$
- Factorial:
  $n! = n * ((n-1)!)$, or
  $0! = 1$

Outline

Recursive programming paradigm

- Recursion in computer science is a self-referential **programming paradigm**, as opposed to iteration with a for() loop, for example.
- Practically, recursion is a process in which a function calls itself.
- Often, the solution to a problem can employ solutions to **smaller instances of the same problem**.

The formal parts

- Recursive case
- Base case (which often results in termination)
- Condition or test, often an if()

Outline

Outline

Example: Exponentiation

- Recursive case:
  $b^n = b * b^{n-1}$
- Base case (termination):
  $b^0 = 1$ (or slightly faster: $b^1 = b$)
- Condition/test, which checks for the base:
  if(n==0):

For example:
$b^4 = b * b^3 = b * (b * b^2) = b * (b * (b * b^1)) = b * (b * (b * (b * b^0)))$
Observe code

Efficient exponentiation

To obtain b^n, do recursively:

- if n is even, do $b^{n/2} * b^{n/2}$
- if n is odd, do $b * b^{n/2} * b^{n/2}$
- with base case, $b^1 = b$

Note: $n/2$ is integer division

What is b^{62} ?

1. $b^{62} = (b^{31})^2$
2. $b^{31} = b(b^{15})^2$
3. $b^{15} = b(b^7)^2$
4. $b^7 = b(b^3)^2$
5. $b^3 = b(b^1)^2$
6. $b^1 = b$

What is b^{61} ?

1. $b^{61} = b(b^{30})^2$
2. $b^{30} = (b^{15})^2$
3. $b^{15} = b(b^7)^2$
4. $b^7 = b(b^3)^2$
5. $b^3 = b(b^1)^2$
6. $b^1 = b$

How many multiplications when counting from the bottom up?

Outline

Paying attention to order is important

- When a function calls itself once, instructions placed before the recursive call are executed once per recursion
- Instructions placed after the recursive call are executed repeatedly after the maximum recursion has been reached.

Observe code

Outline

Example: Factorial

- Recursive case:
  $n! = n * ((n-1)!)$
- Base case (termination):
  $0! = 1$
- Condition/test, which checks for the base:
  if(n==0):

Observe code

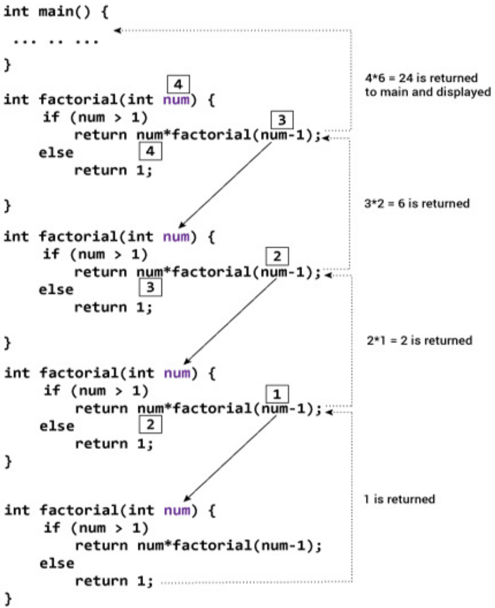
Example: Factorial

$n! = n * ((n-1)!)$
$0! = 1$

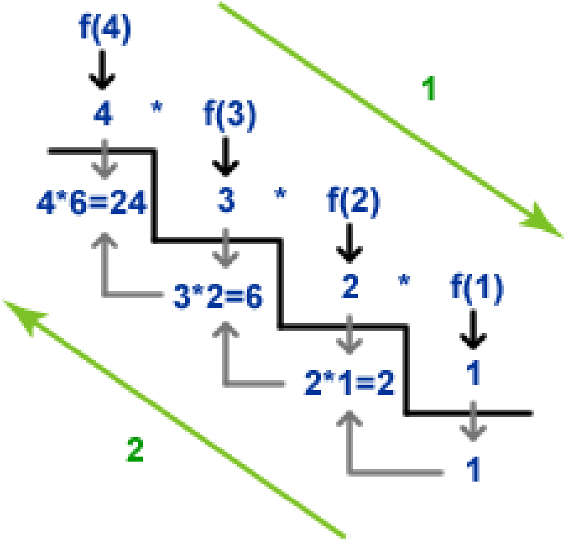
Observe code, and evaluate calling factorial of 3:

```
Since 3 is not 0, take second branch and calculate (n-1)!...
        Since 2 is not 0, take second branch and calculate (n-1)!...
                Since 1 is not 0, take second branch and calculate (n-1)!...
                        Since 0 equals 0, take first branch and return 1.
                Return value, 1, is multiplied by n = 1, and return result.
        Return value, 1, is multiplied by n = 2, and return result
Return value, 2, is multiplied by n = 3, and the result, 6, becomes the
return value of the function call that started the whole process.
```

Factorial


int main() {
... .
}
int factorial(int num) {
if (num > 1)
return num*factorial(num-1);
else
return 1;
}
4
3
4
2
3
1
2
4*6 = 24 is returned
to main and displayed
3*2 = 6 is returned
2*1 = 2 is returned
1 is returned

Factorial



Factorial

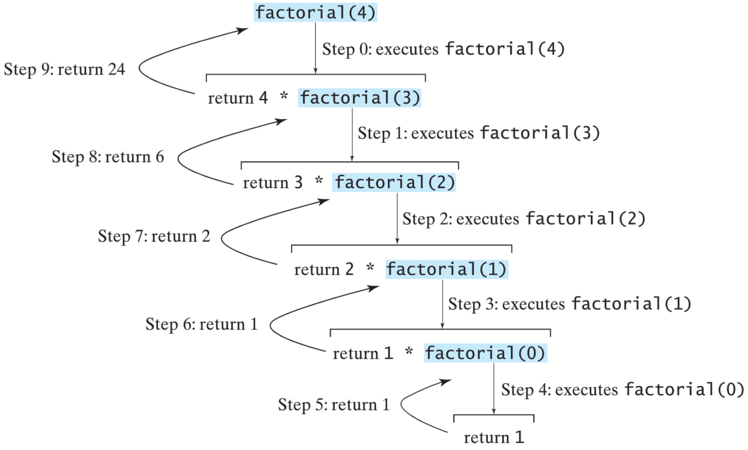


FIGURE 18.2 Invoking `factorial(4)` spawns recursive calls to `factorial`.

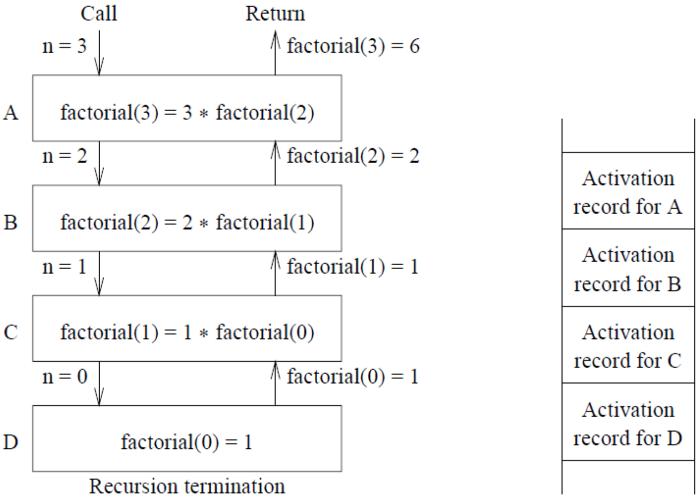
Outline

Keeping the stack of activation records

- A subroutine call is implemented by placing necessary information about the subroutine (including the return address, parameters, and local variables) onto a stack. This information is called an activation record.
- Further subroutine calls add to the stack. Each return from a subroutine pops the top activation record off the stack.

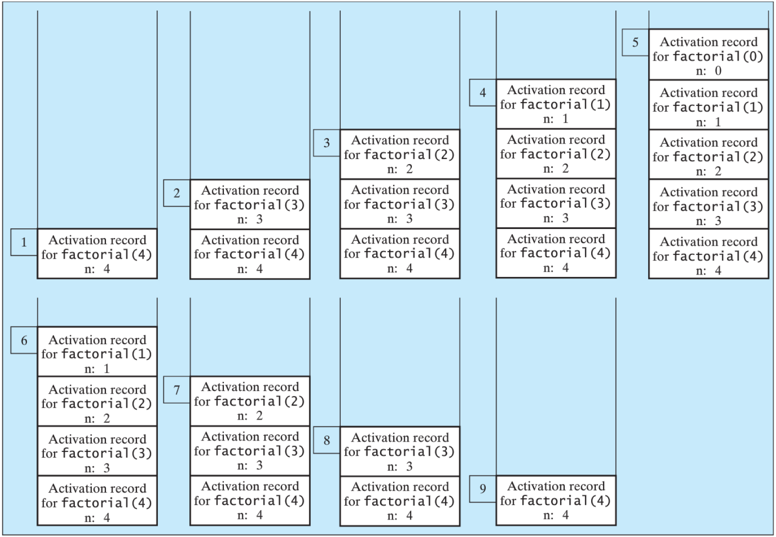
Outline

Keeping the stack of activation records

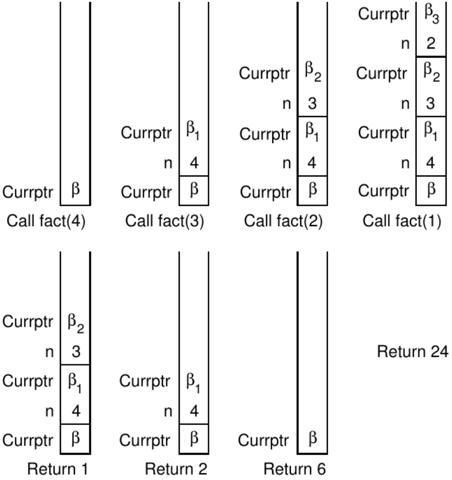


(a) (b)

Keeping the stack of activation records



Keeping the stack of activation records



Each record on the stack needs a return address (more later)

Outline

Activation records

In practice, a recursive call doesn't make an entire new copy of a routine. Each function call gets an activation record where it:

1. Records the location to which it will return
2. Re-enters the new function code at the beginning
3. Allocates memory for the local data for this new invocation of the function
4. ... until the base case, then:
5. Returns to the earlier function right after where it left off...

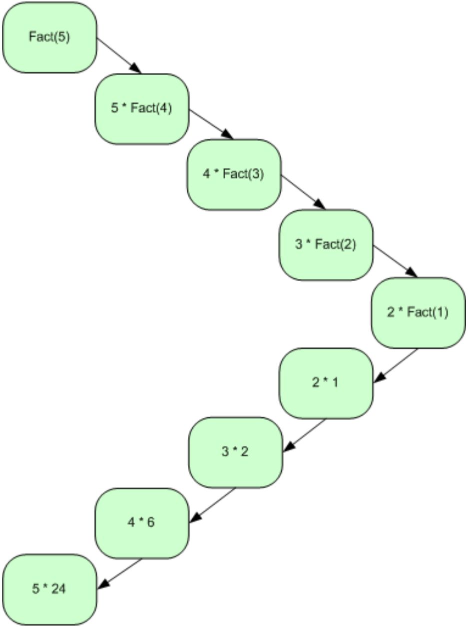
Outline

Recursive programming variations: overview

- **Single recursion** contains a single self-reference, e.g., list traversal, linear search, or computing the factorial function... Single can often be replaced by an iterative computation, running in linear time and requiring constant space.
- **Multiple recursion (binary included)** contains multiple self-references, e.g., tree traversal, depth-first search (coming up)... This may require exponential time and space, and is more fundamentally recursive, not being able to be replaced by iteration without an explicit stack.
- **Indirect (or mutual) recursion** occurs when a function is called not by itself but by another function that it called (either directly or indirectly). Chains of three or more functions are possible.
- **Generative recursion** acts on outputs it generated, while **structural recursion** acts on progressive sets of input data.

Outline

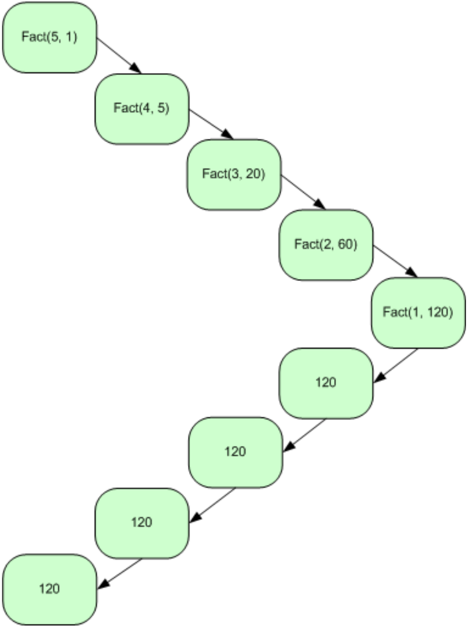
Linear recursion



$$f(n) = \begin{cases} 1 & n = 0 \\ n * f(n-1) & n > 0 \end{cases}$$

Outline

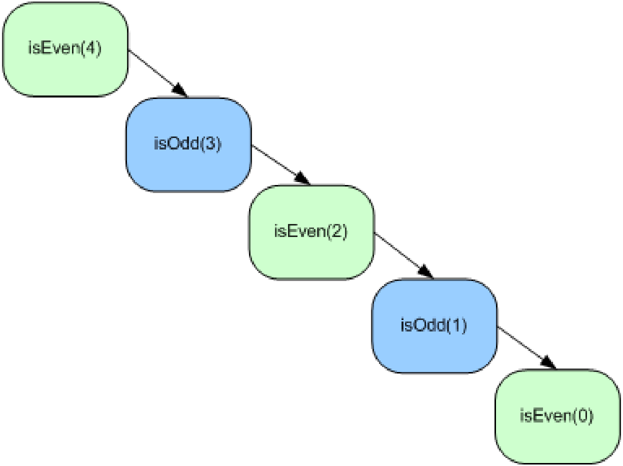
Tail recursion



$$f(n,a)=\begin{cases} a & n=0 \\ f(n-1,n*a) & n>0 \end{cases}$$

Outline

Mutual recursion



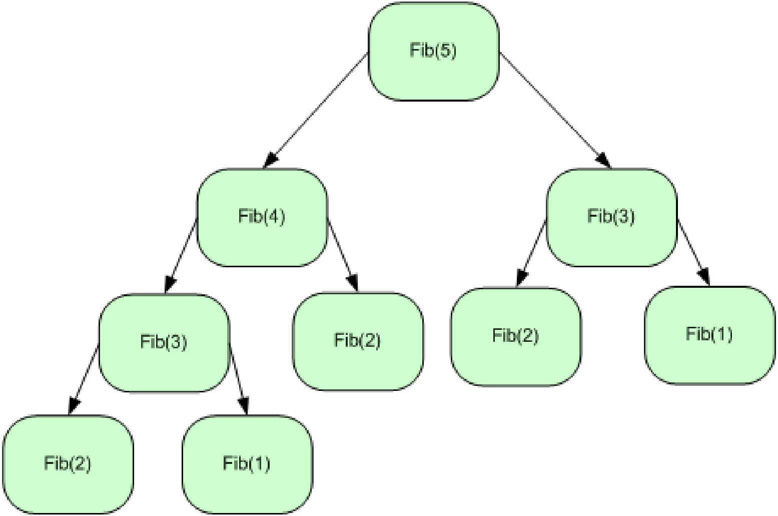
$$isEven(n) = \begin{cases} true & n = 0 \\ isOdd\,(n-1) & n > 0 \end{cases}$$

$$isOdd\,(n) = \begin{cases} false & n = 0 \\ isEven(n-1) & n > 0 \end{cases}$$

Outline

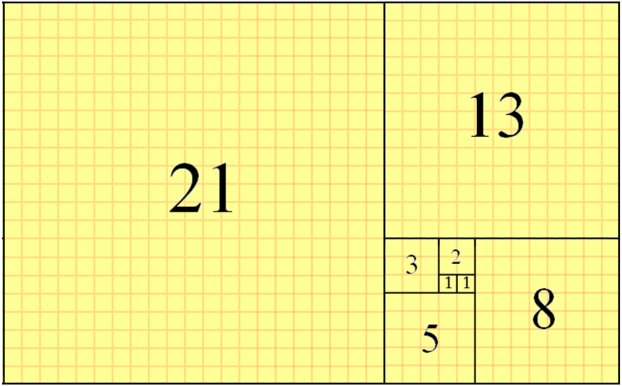
Binary (a kind of multiple) recursion

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$



Fibonacci series

Recursive case: $F_n = F_{n-1} + F_{n-2}$
Base case: $F_0 = 0, F_1 = 1$
1,1,2,3,5,8,13,...



Check out the code