RECURSION

ALGORITHMS & DATA STRUCTURES – I

COMP 221

Recursion

A very old idea, with its roots in mathematical induction. It always has:

- An anchor (or base or trivial) case
- An inductive case

So, a function is said to be defined recursively if its definition consists of

- An **anchor or base case** in which the function's value is defined for one or more values of the parameters
- An **inductive or recursive step** in which the function's value (or action) for the current parameter values is defined in terms of previously defined function values (or actions) and/or parameter values.

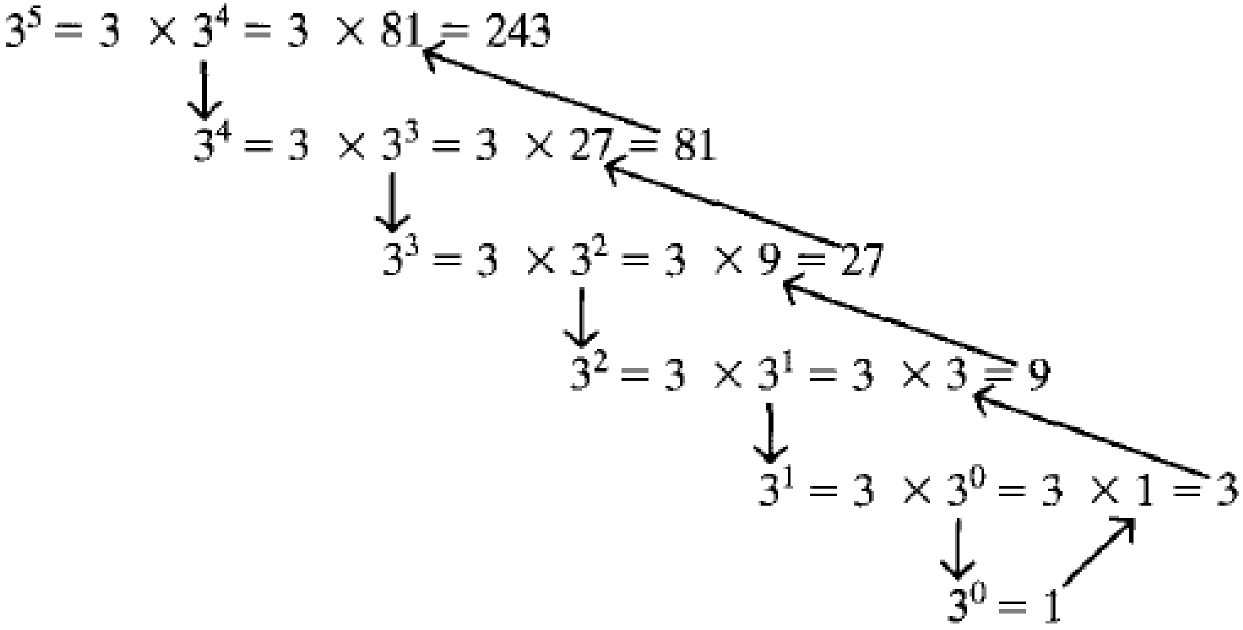
Example: Power Function

- $x^0 = 1$ (the anchor or base case)
- For $n > 0$ $x^n = x * x^{n-1}$ *(the inductive or recursive case)*

$$\begin{tikzcd}
3^5 = 3 \times 3^4 \arrow[dr] & & & & & \\
& 3^4 = 3 \times 3^3 \arrow[dr] & & & & \\
& & 3^3 = 3 \times 3^2 \arrow[dr] & & & \\
& & & 3^2 = 3 \times 3^1 \arrow[dr] & & \\
& & & & 3^1 = 3 \times 3^0 \arrow[dr] & \\
& & & & & 3^0 = 1
\end{tikzcd}$$

Power Function

Since the value of 30 is given, we can now backtrack to find the value of 3^1 and so on.



$3^5 = 3 \times 3^4 = 3 \times 81 = 243$

$3^4 = 3 \times 3^3 = 3 \times 27 = 81$

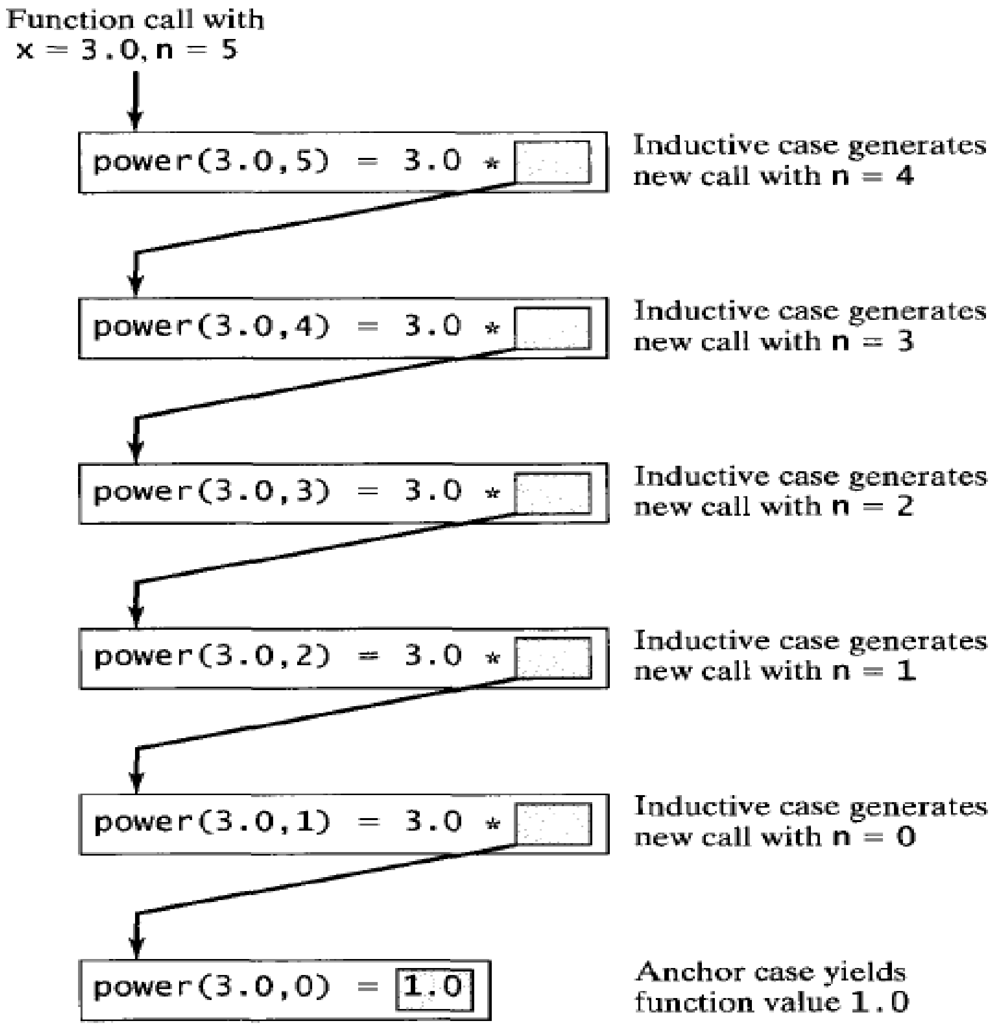
$3^3 = 3 \times 3^2 = 3 \times 9 = 27$

$3^2 = 3 \times 3^1 = 3 \times 3 = 9$

$3^1 = 3 \times 3^0 = 3 \times 1 = 3$

$3^0 = 1$

Recursive Calls to the Power Function



Function call with
x = 3.0, n = 5

`power(3.0,5) = 3.0 *` ☐ Inductive case generates new call with **n** = 4

`power(3.0,4) = 3.0 *` ☐ Inductive case generates new call with **n** = 3

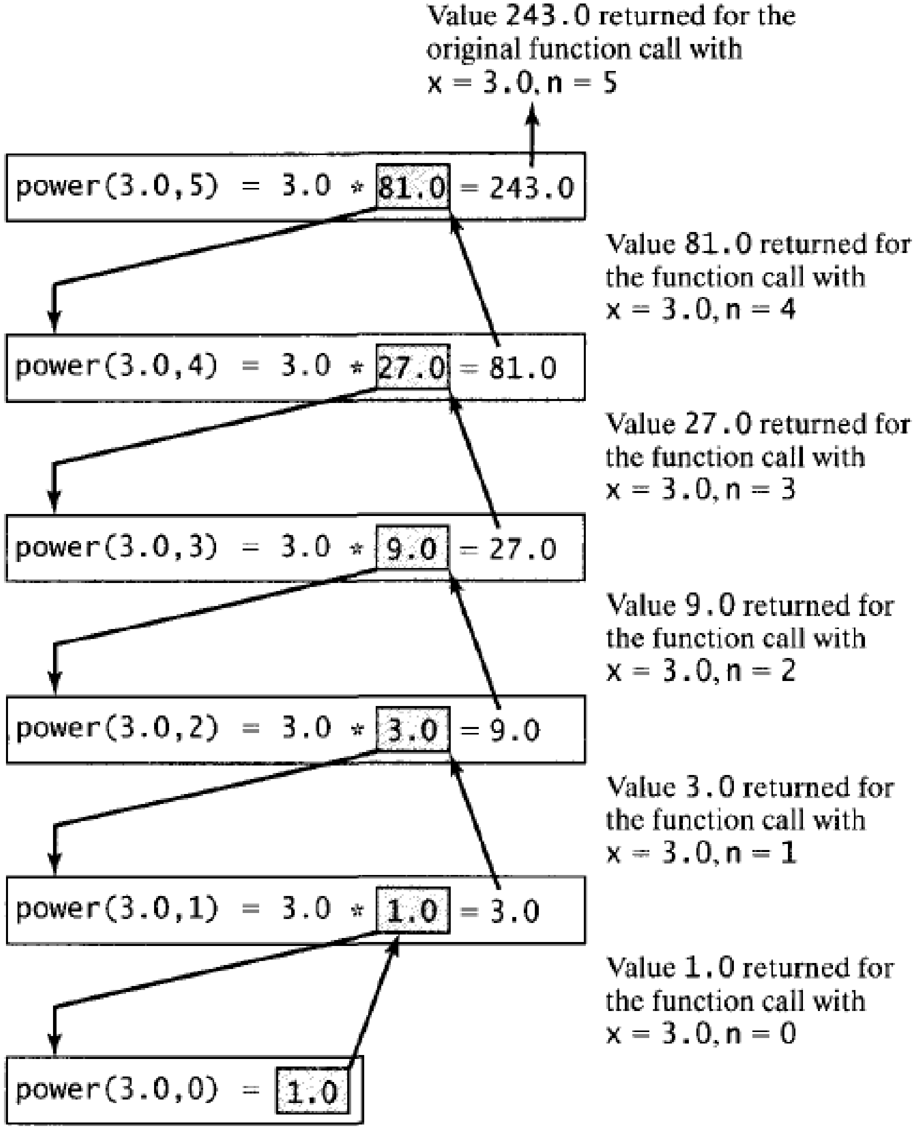
`power(3.0,3) = 3.0 *` ☐ Inductive case generates new call with **n** = 2

`power(3.0,2) = 3.0 *` ☐ Inductive case generates new call with **n** = 1

`power(3.0,1) = 3.0 *` ☐ Inductive case generates new call with **n** = 0

`power(3.0,0) = 1.0` Anchor case yields function value `1.0`

Recursive Calls to the Power Function



Value 243.0 returned for the original function call with x = 3.0, n = 5

power(3.0,5) = 3.0 * 81.0 = 243.0

Value 81.0 returned for the function call with x = 3.0, n = 4

power(3.0,4) = 3.0 * 27.0 = 81.0

Value 27.0 returned for the function call with x = 3.0, n = 3

power(3.0,3) = 3.0 * 9.0 = 27.0

Value 9.0 returned for the function call with x = 3.0, n = 2

power(3.0,2) = 3.0 * 3.0 = 9.0

Value 3.0 returned for the function call with x = 3.0, n = 1

power(3.0,1) = 3.0 * 1.0 = 3.0

Value 1.0 returned for the function call with x = 3.0, n = 0

power(3.0,0) = 1.0

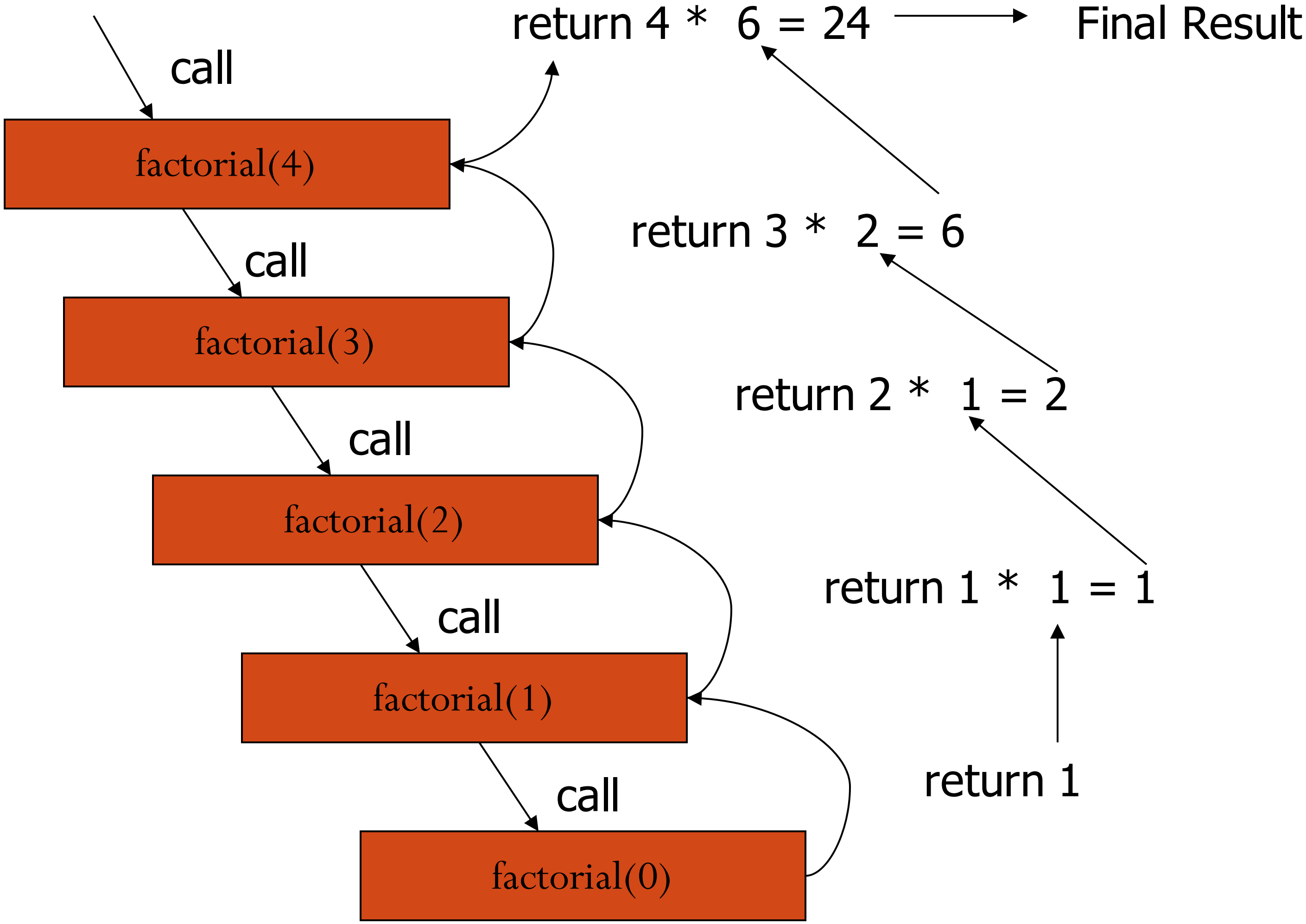
Example: Factorials

- Base case: 0! = 1
- Inductive case: n! = n*(n-1)!

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

T(n) = O(n)

Factorial



call

factorial(4)

call

factorial(3)

call

factorial(2)

call

factorial(1)

call

factorial(0)

return 1

return 1 * 1 = 1

return 2 * 1 = 2

return 3 * 2 = 6

return 4 * 6 = 24

Final Result

Comments on Recursion

However, many common textbook examples of recursion are *tail-recursive*, i.e. the last statement in the recursive function is a recursive invocation.

Tail-recursive functions can be written (much) more efficiently using a loop.

Tail recursive functions are often said to “return the value of the last recursive call as the value of the function”

Binary Recursion

- When an algorithm makes two recursive calls we say that it uses binary recursion.
- Fibonacci numbers are recursively defined as follows:

$F_0 = 0$

$F_1 = 1;$

$F_i = F_{i-1} + F_{i-2}$ for $i > 1$

Fibonacci Numbers

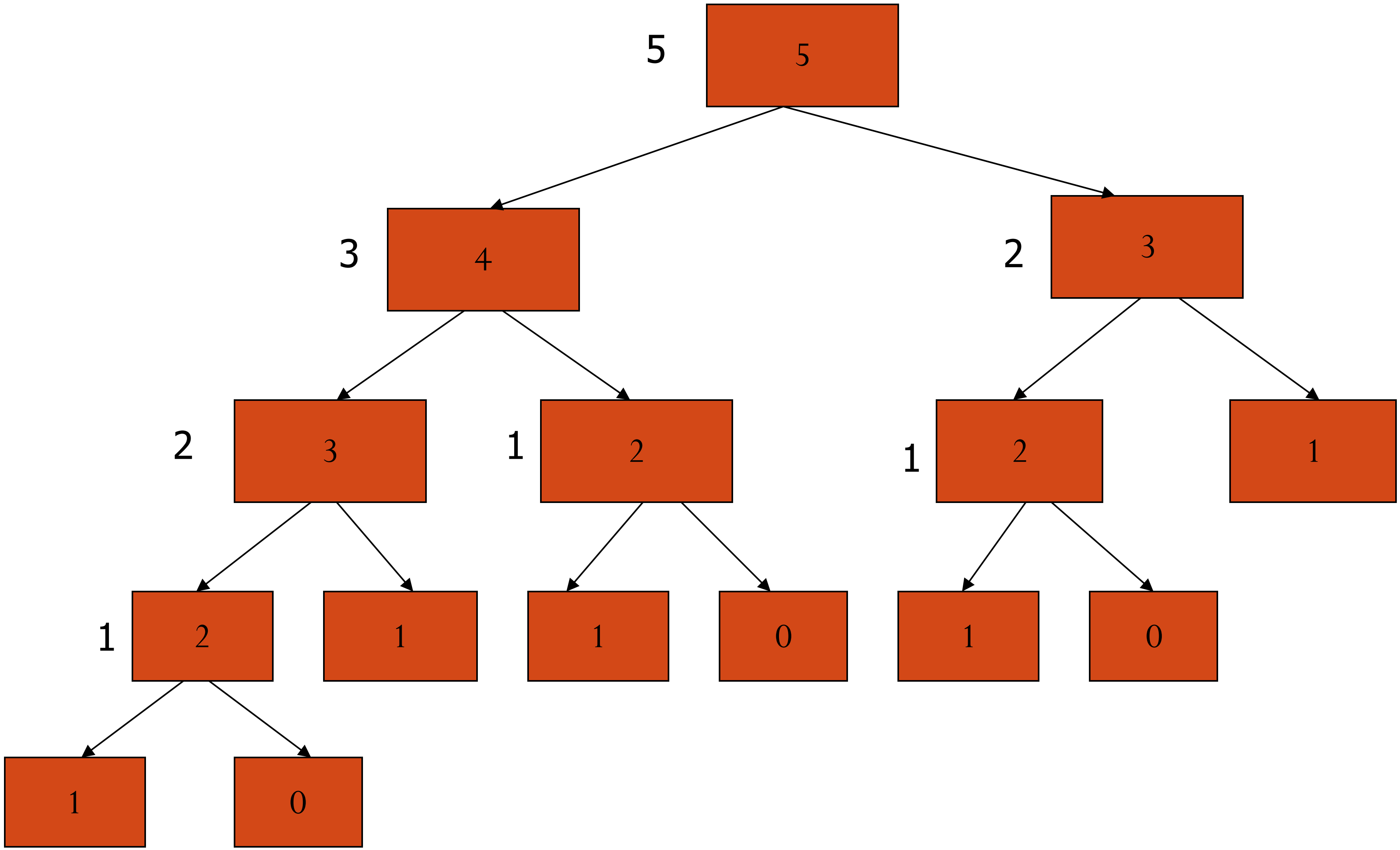
Algorithm Fib(n)

Input: Nonnegative integer n.

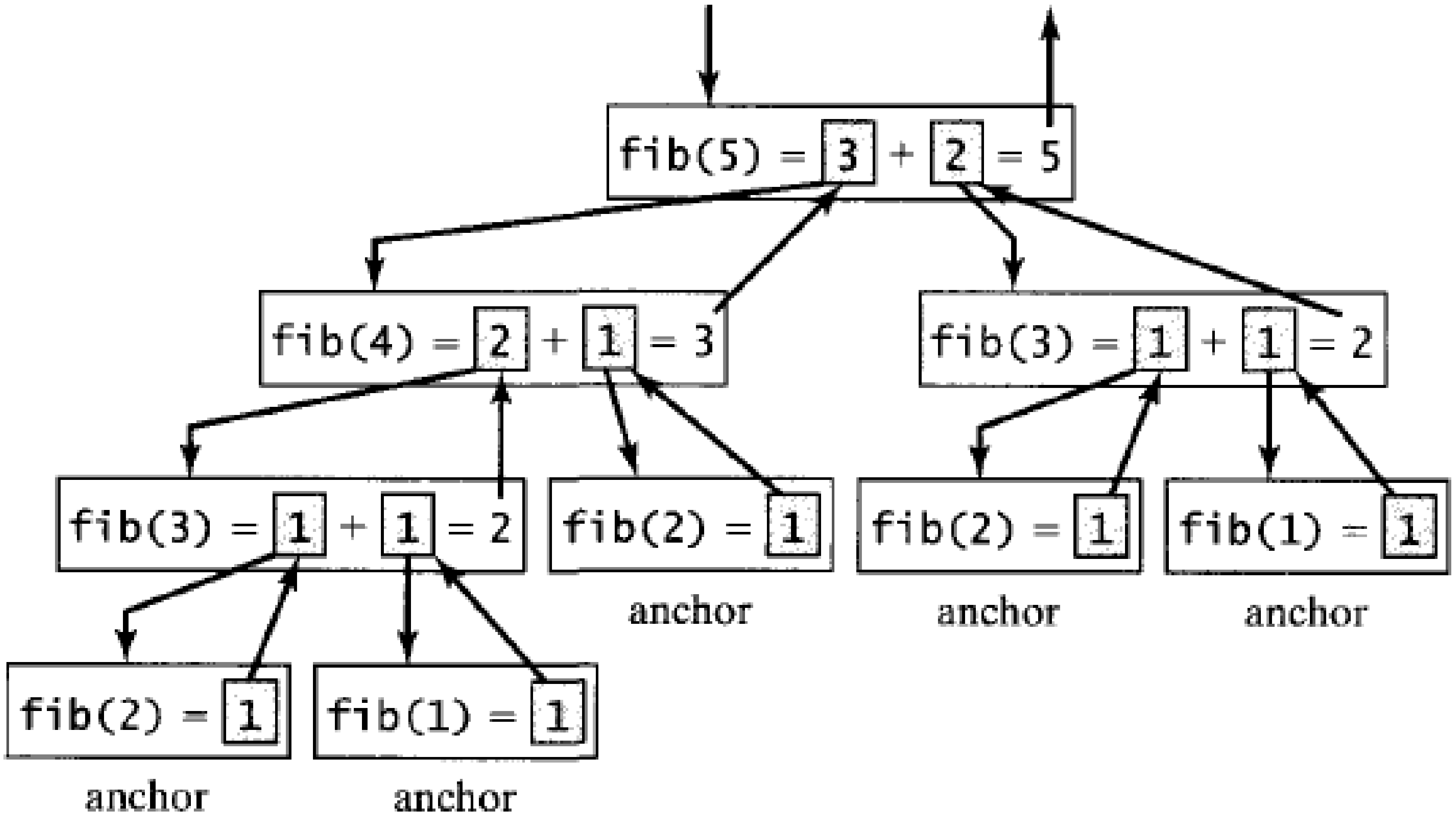
Output: The nth Fibonacci number F_n.

```
unsigned fib(unsigned n)
{
if (n <= 2)
return 1; // anchor case
// else
return fib(n - 1) + fib(n - 2); // inductive case (n > 2)
}
```

Recursive Trace



Recursion Tree



```
// Fibonacci numbers
int F(unsigned n)
{                                  // recursive, expensive!
  if (n < 3)
    return 1;
  else
    return F(n -1) + F(n - 2);
                               }
```

Recursive: $O(1.7^n)$

```
int F(unsigned n)
{                                  // iterative, cheaper
  int fib1 = 1, fib2 = 1;
  for (int i = 3; i <= n; i++)
  {
    int fib3 = fib1 + fib2;
    fib1 = fib2;
    fib2 = fib3;
  }
  return fib2;
}
```

Iterative: $O(n)$

More complicated recursive functions are sometimes replaced by iterative functions that use a stack to store the recursive calls. (See Section 7.3)

```
// Counting the number of digits in a positive integer
int F(int n, int count)
{                                 // recursive, expensive!
  if (n < 10)
    return 1 + count;
  else
    return F(n/10, ++count);
                               }

int F(int n)
{                                 // iterative, cheaper
  int count = 1;
  while (n >= 10)
  {
    count++;
    n /= 10;
  }
  return count;
}
```

Computing times of recursive functions

Have to solve a recurrence relation.

In emacs/xemacs:
Esc-n Esc-x hanoi

```
//  Towers of Hanoi
void Move(int n, char source, char destination, char spare)
{
if (n <= 1)                  // anchor (base) case
  cout << "Move the top disk from " << source
        << " to " << destination << endl;
 else
 {                           // inductive case
   Move(n-1, source, spare, destination);
   Move(1, source, destination, spare);
   Move(n-1, spare, destination, source);
 }
}
```

T(n) = $O(2^n)$

THANK YOU

?