Lecture 9
Recursion

Recursion

- The best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type
- A procedure that is defined in terms of itself
- A function that calls itself is known as a recursive function. And, this technique is known as recursion.

Example

- The factorial function(Iterative)

  6! = 6 * 5 * 4 * 3 * 2 * 1

  5! = 5 * 4 * 3 * 2 * 1

- We could write: (Recursive)

  6! = 6 * 5!

  6! = 6*(6-1)!

Recursive Function

- *Recursive Function:–* a function that calls itself
    - Directly or indirectly

- Each recursive call is made with a new, independent set of arguments
    - Previous calls are suspended

Recursion

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

Example 1: The Factorial function

We can express the factorial function as follows:

```
n! = n * (n-1)!
```

More precisely,

```
n! = 1          {if n is equal to 0}
n! = n*(n-1)!   {if n is larger than 0}
```

The C code of factorial function:

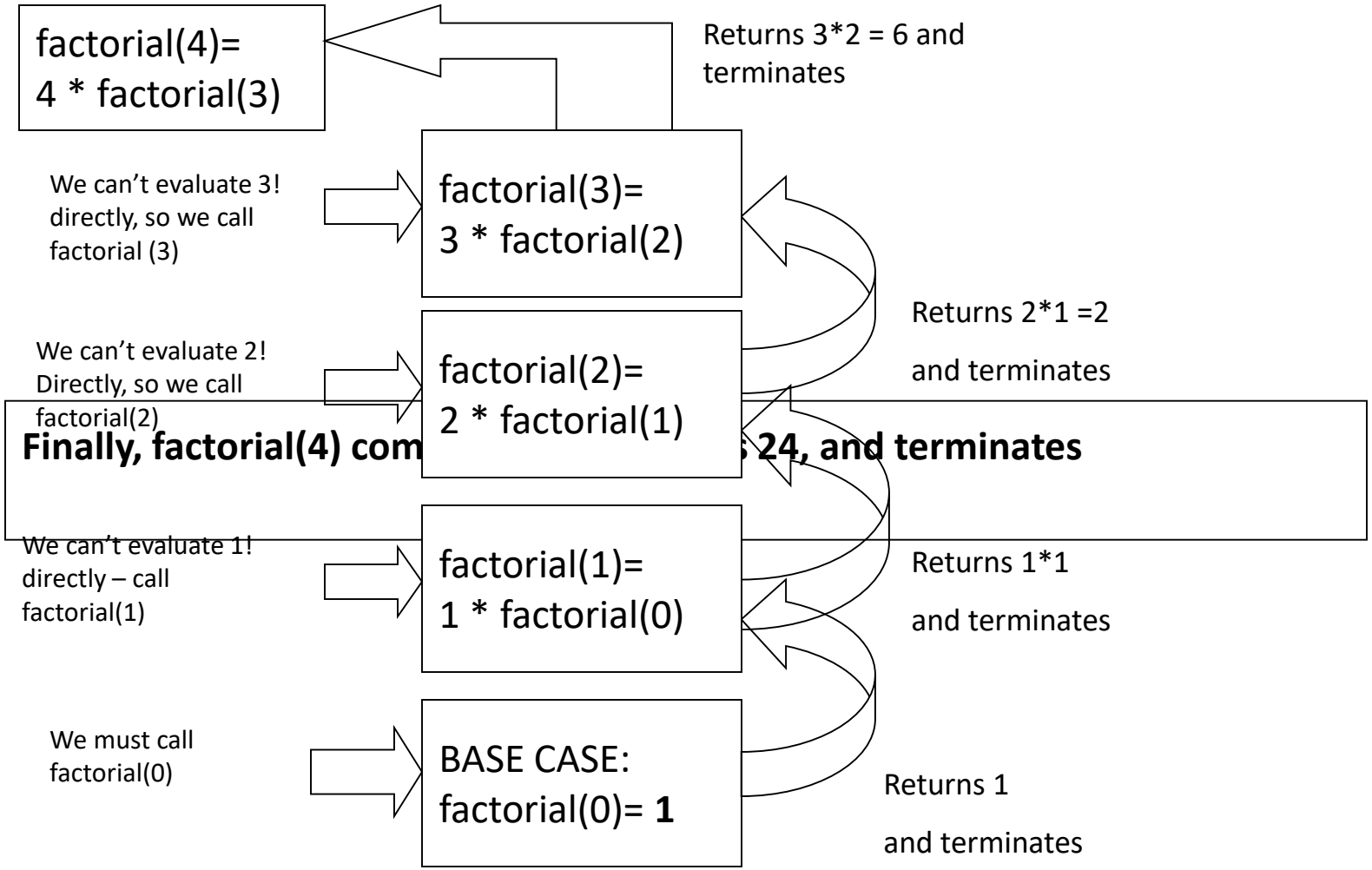
```
int factorial (int n)
{
  //base case
  if (n ==0) return 1;

  //recursive call
  return n * factorial (n – 1);
}
```

Content of a Recursive Method

- **Base case(s).**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.
- **Recursive calls.**
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Trace of a call to Factorial: int v = factorial(4)



Example 2: Fibonacci Number Sequence

`1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

where each number is the sum of the preceding two.

- Recursive definition for the nth Fibonacci number:
  - **`F(1) = 1;`**
  - **`F(2) = 1;`**
  - **`F(nth) = F(nth-1)+ F(nth-2)`**

The C code of nth Fibonacci number:

```
int fib(int nth)
{
//Base Case
 if (nth == 1) return 1;
 if (nth == 2) return 1;

// recurrence calls
 return (fib(nth-1)+fib(nth-2));
}
```

Tracing with Multiple Recursive Calls

Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(5):	Fib returns Fib(3) + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(1):	Fib returns 1
Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(2):	Fib returns 1
Fib(3):	Fib returns 1 + Fib(2)
Fib(5):	Fib returns Fib(3) + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + 1
Fib(5):	Fib returns Fib(3) + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(4):	Fib returns Fib(2) + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(2):	Fib returns 1
Fib(4):	Fib returns Fib(2) + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(1):	Fib returns 1
Fib(3):	Fib returns Fib(1) + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(2):	Fib returns 1
Fib(3):	Fib returns 1 + Fib(2)
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(3):	Fib returns 1 + 1
Fib(4):	Fib returns 1 + Fib(3)
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Fib(4):	Fib returns 1 + 2
Fib(5):	Fib returns 2 + Fib(4)
Main Algorithm:	answer <- Fib(5)

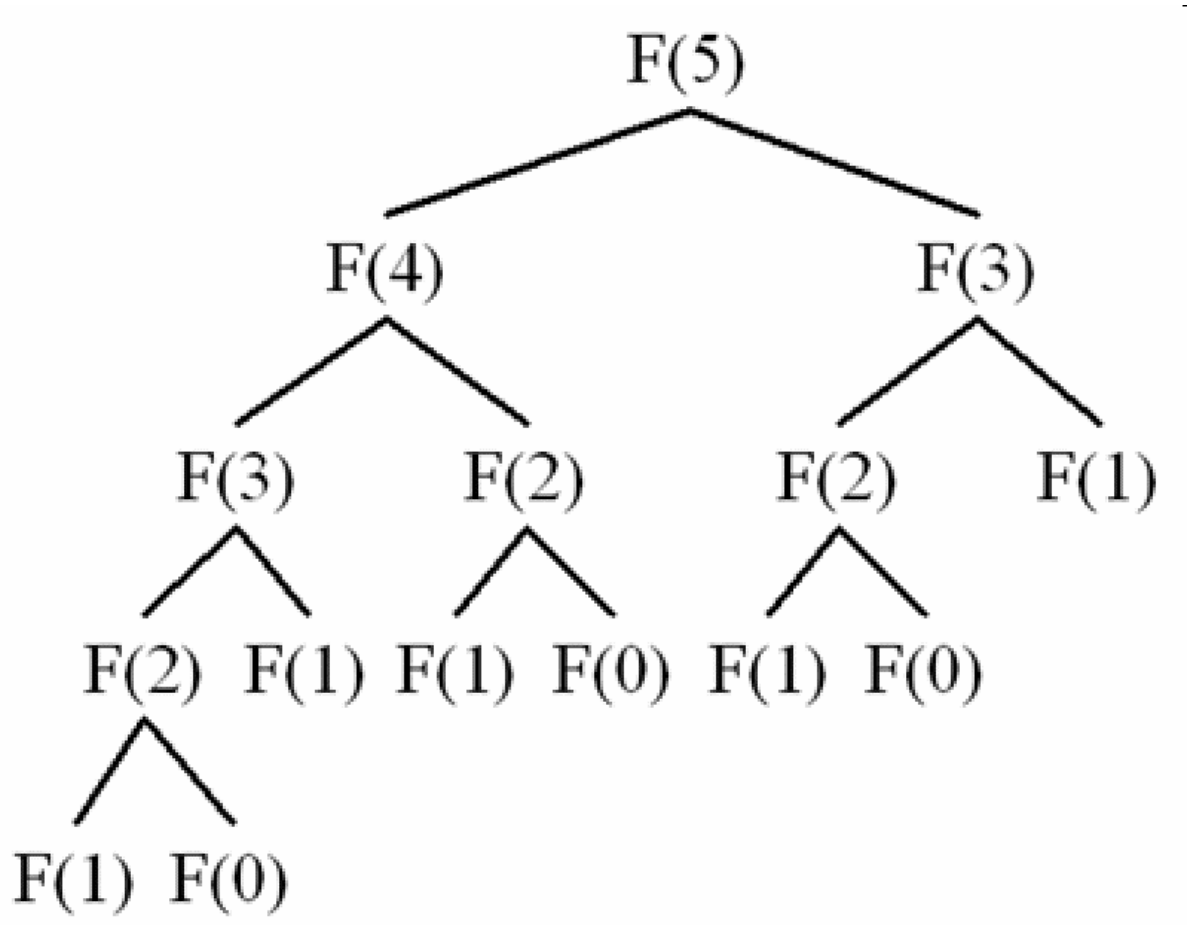
Tracing with Multiple Recursive Calls

Fib(5):	Fib returns 2 + 3
Main Algorithm:	answer <- Fib(5)

Tracing with Multiple Recursive Calls

Main Algorithm:	answer <- 5

Fibonacci number (Tree Structure)



Advantages

- Reduce code length.
- Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex.

Disadvantages

- Recursive solution is always logical and it is very difficult to trace.(debug and understand).
- In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
- Recursion takes a lot of stack space.
- Recursion uses more processor time.