



THE UNIVERSITY OF VERMONT  
COLLEGE OF ENGINEERING &  
MATHEMATICAL SCIENCES

# RECURSION

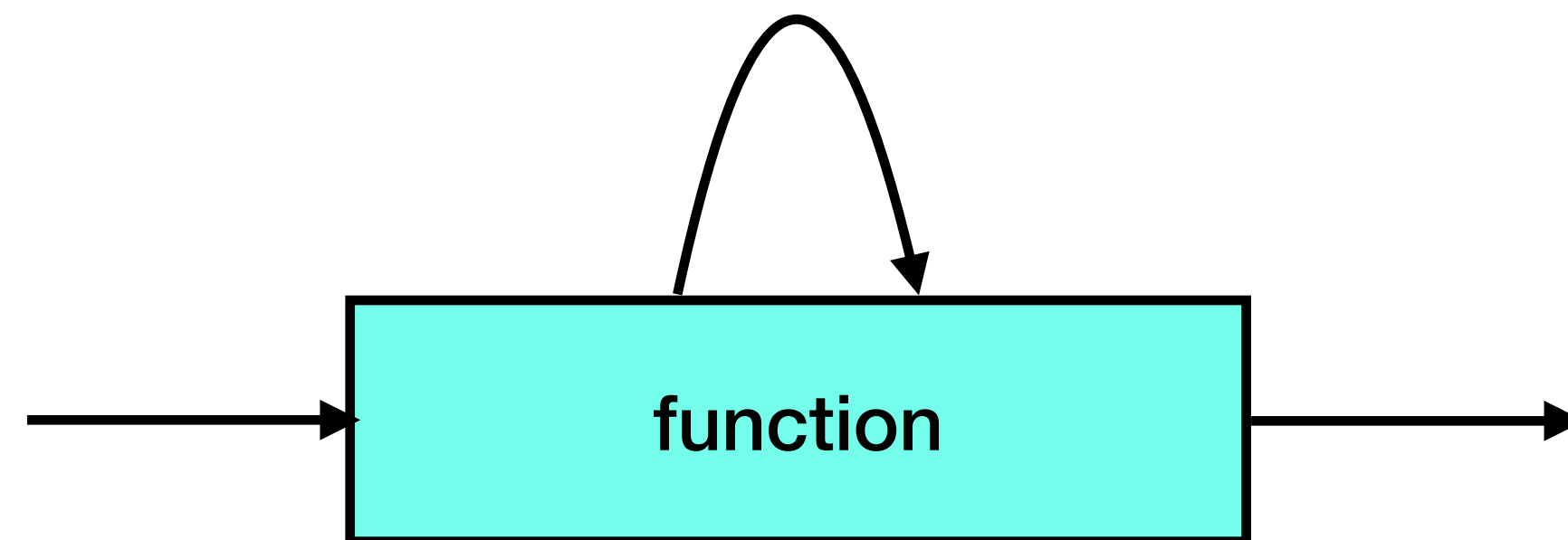
## AN INTRODUCTION

CS 124 / Department of Computer Science

# What is recursion?

You may think of *recursion* as a programming structure where a function calls itself. We call such a function a *recursive function*.

Many algorithms can be implemented using recursion.



# What is recursion good for?

- Recursion can provide an elegant solution which breaks a problem down into smaller parts.
- Recursion is used in numeric calculations, tree traversals, and many other applications.
- Recursion can solve problems without requiring an explicit loop.

# The factorial function

A classic example of a recursive function is the *factorial function*, where we calculate the value of  $n!$  using recursion. Recall that the factorial function  $n!$  is defined as

$$n! = \prod_{i=1}^n i$$

For example,  $5! = 1 \times 2 \times 3 \times 4 \times 5$ .

$0!$  is defined to be 1.

# Recursive function to calculate n!

```
int factorial(int n) {  
    if ((n == 0) || (n == 1)) {  
        // recall 0! is defined to be 1  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

# Recursive function to calculate n!

```
int factorial(int n) {  
    if ((n == 0) || (n == 1)) {  
        // recall 0! is defined to be 1  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

Here's the recursion



# Recursive function to calculate n!

```
int factorial(int n) {
```

```
    if ((n == 0) || (n == 1)) {
```

```
        // recall 0! is defined to be 1
```

```
        return 1;
```

```
    } else {
```

```
        return n * factorial(n - 1);
```

```
    }
```

```
}
```

← What's all this?

# Recursive function to calculate n! (incorrect)

```
int factorial(int n) {  
    return n * factorial(n - 1);  
}
```

What happens if we call this function?



# Recursive function to calculate n! (incorrect)

```
int factorial(int n) {  
    return n * factorial(n - 1);  
}
```

Let's say we call this function, providing the value 5 as an input parameter.

This will calculate  $5 \times 4 \times 3 \times 2 \times 1 \times 0 \times -1 \times -2 \times -3 \times -4 \dots$  etc.

# Recursive function to calculate n! (incorrect)

```
int factorial(int n) {  
    return n * factorial(n - 1);  
}
```


Let's say we call this function, providing the value 5 as an input parameter.

This will calculate  $5 \times 4 \times 3 \times 2 \times 1 \times 0 \times -1 \times -2 \times -3 \times -4 \dots$  etc.

This will *never terminate!*

# Recursive function to calculate n!

```
int factorial(int n) {  
    if ((n == 0) || (n == 1)) {  
        // recall 0! is defined to be 1  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```



Base cases

In addition to the recursive case, a recursive function must have one (or more) *base cases*, that provide termination criteria for the function.

# The parts of a recursive function

A recursive function must have

- a *recursive case*, and
- one or more *base cases* (without recursion)

The recursive case breaks the problem down into smaller instances. The base cases provide termination criteria, breaking the chain of recursion and preventing infinite regress.

# The Fibonacci function

Another classic example of recursion is the Fibonacci function.

The Fibonacci sequence starts with 0, 1, and then calculates subsequent terms by taking the sum of the previous two terms in the sequence. So a Fibonacci number, denoted  $F_n$  where  $n$  is an index, is calculated thus

$$\begin{cases} F_n = 0 & n = 0 \\ F_n = 1 & n = 1 \\ F_n = F_{n-2} + F_{n-1} & \textit{otherwise} \end{cases}$$

# The Fibonacci function

Another classic example of recursion is the Fibonacci function.

A Fibonacci sequence starts with 0, 1, and then calculates subsequent terms by taking the sum of the previous two terms in the sequence. So a Fibonacci number, denoted  $F_n$  where  $n$  is an index, is calculated thus

$$\begin{cases} F_n = 0 & n = 0 \\ F_n = 1 & n = 1 \\ F_n = F_{n-2} + F_{n-1} & \textit{otherwise} \end{cases} \quad \leftarrow \text{Base cases}$$

# The Fibonacci function

Another classic example of recursion is the Fibonacci function.

A Fibonacci sequence starts with 0, 1, and then calculates subsequent terms by taking the sum of the previous two terms in the sequence. So a Fibonacci number, denoted  $F_n$  where  $n$  is an index, is calculated thus

$$\begin{cases} F_n = 0 & n = 0 \\ F_n = 1 & n = 1 \\ F_n = F_{n-2} + F_{n-1} & \textit{otherwise} \end{cases}$$

← The recursive case

# The Fibonacci sequence

Using the definition given, we may calculate the Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...



# Fibonacci in C++

Here's the Fibonacci function to return the  $n^{\text{th}}$  Fibonacci number, implemented in C++:

```
int fibonacci(int x) {  
    if (x == 0) || (x == 1) {  
        return(x);  
    } else {  
        return(fibonacci(x - 2) + fibonacci(x - 1));  
    }  
}
```

# Fibonacci in C++

Here's the Fibonacci function to return the  $n^{\text{th}}$  Fibonacci number, implemented in C++:

```
int fibonacci(int x) {  
    if (x == 0) || (x == 1) {  
        return(x);  
    } else {  
        return(fibonacci(x - 2) + fibonacci(x - 1));  
    }  
}
```

# Fibonacci in C++

Here's the Fibonacci function to return the  $n^{\text{th}}$  Fibonacci number, implemented in C++:

```
int fibonacci(int x) {  
    if (x == 0) || (x == 1) {  
        return(x);  
    } else {  
        return(fibonacci(x - 2) + fibonacci(x - 1));  
    }  
}
```

# What's the downside?

- If not appropriate to a particular problem or poorly implemented, recursion can be inefficient and slow.
- For example, while factorial and Fibonacci are convenient examples for demonstrating recursion, neither of the implementations given is efficient. (But they serve their purpose here.)

# Summary

- Recursion is an elegant approach to break a problem down into smaller instances and solve by recurring calculation.
- Recursive functions are functions that call themselves.
- Recursive functions require a recursive case and at least one base case.
- While elegant, they may not be the most efficient approach, so use with caution.