Recursive functions

Michael Nowak

Texas A&M University

Overview

Overview

Basics of recursion

- In mathematics, a `recursive function` is a function that is defined in terms of itself

Basics of recursion

- In mathematics, a `recursive function` is a function that is defined in terms of itself
- From our function basics lecture, we introduced the factorial function

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

Basics of recursion

- In mathematics, a `recursive function` is a function that is defined in terms of itself
- From our function basics lecture, we introduced the factorial function
$$n! = \prod_{i=1}^{n} i$$
- This definition assumes that we know how to make the multiplication happen repeatedly

Basics of recursion

- In mathematics, a `recursive function` is a function that is defined in terms of itself
- From our function basics lecture, we introduced the factorial function

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

- This definition assumes that we know how to make the multiplication happen repeatedly
- We can make the repetitive multiplication more explicit by writing the definition of this function using recursion

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

Basics of recursion

- In mathematics, a `recursive function` is a function that is defined in terms of itself
- From our function basics lecture, we introduced the factorial function
$$n! = \prod_{i=1}^{n} i$$
- This definition assumes that we know how to make the multiplication happen repeatedly
- We can make the repetitive multiplication more explicit by writing the definition of this function using recursion
$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$
  - Using this definition, we are defining factorial in terms of factorial

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`
    - All recursive functions need a `base case`

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`
    - All recursive functions need a `base case`
    - The defining attribute of the `base case` is that it is *not* recursive

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`
    - All recursive functions need a `base case`
    - The defining attribute of the `base case` is that it is *not* recursive
    - Without a `base case`, you'd get `infinite recursion`

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`
    - All recursive functions need a `base case`
    - The defining attribute of the `base case` is that it is *not* recursive
    - Without a `base case`, you'd get `infinite recursion`
  - The other necessary case is the `recursive case`

Basics of recursion

- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`
    - All recursive functions need a `base case`
    - The defining attribute of the `base case` is that it is *not* recursive
    - Without a `base case`, you'd get `infinite recursion`
  - The other necessary case is the `recursive case`
    - The `recursive call` is made with a value that moves the `recursive function` towards its `base case`

Basics of recursion

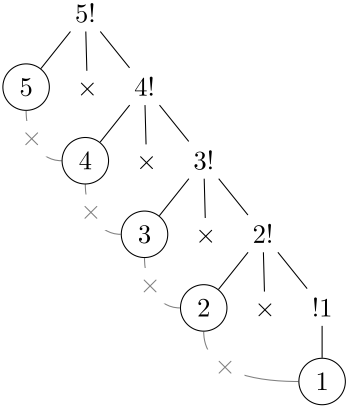
- It is apparent in our recursive definition of factorial that there are two cases:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

  - In the case where $n < 2$, factorial evaluates to 1; this is the `base case`
    - All recursive functions need a `base case`
    - The defining attribute of the `base case` is that it is *not* recursive
    - Without a `base case`, you'd get `infinite recursion`
  - The other necessary case is the `recursive case`
    - The `recursive call` is made with a value that moves the `recursive function` towards its `base case`
    - In this case, we define $n!$ as $(n-1)!$

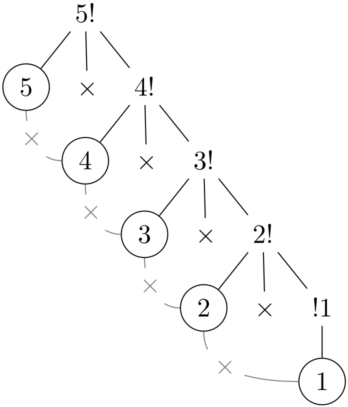
Basics of recursion

- Using our recursive definition of factorial, we would solve 5! as:



Basics of recursion

- Using our recursive definition of factorial, we would solve 5! as:



- That is, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Overview

Writing a recursive function

- In the function basics lecture, we wrote an iterative solution for

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

as

Writing a recursive function

- In the function basics lecture, we wrote an iterative solution for

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

as

```
int fact(int val)
{
    int res = 1;
    while(val > 1) {
        res *= val;
        val -= 1;
    }
    return res;
}
```

Writing a recursive function

- We would now like to write a recursive function that calculates the `factorial` of a number

Writing a recursive function

- We would now like to write a recursive function that calculates the `factorial` of a number
- The recursive definition gives us some insight as to how we should go about this:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

Writing a recursive function

- We would now like to write a recursive function that calculates the `factorial` of a number
- The recursive definition gives us some insight as to how we should go about this:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

- Our recursive function `fact` requires:

Writing a recursive function

- We would now like to write a recursive function that calculates the `factorial` of a number
- The recursive definition gives us some insight as to how we should go about this:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

- Our recursive function `fact` requires:
  - One `int`eger parameter, `n`, whose argument we will calculate the factorial for

Writing a recursive function

- We would now like to write a recursive function that calculates the `factorial` of a number
- The recursive definition gives us some insight as to how we should go about this:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

- Our recursive function `fact` requires:
  - One `integer` parameter, `n`, whose argument we will calculate the factorial for
  - A base case, such that when $n < 2$ we `return 1`

Writing a recursive function

- We would now like to write a recursive function that calculates the `factorial` of a number
- The recursive definition gives us some insight as to how we should go about this:

$$n! = \begin{cases} 1 & \text{if } n < 2, \\ n \times (n-1)! & \textit{otherwise} \end{cases}$$

- Our recursive function `fact` requires:
  - One `integer` parameter, `n`, whose argument we will calculate the factorial for
  - A base case, such that when $n < 2$ we `return 1`
  - A recursive case, that moves towards the base case, such that when $n \geq 2$ we `return n*fact(n-1)`

Writing a recursive function

- Our recursive function `fact` requires:
  - One `integer` parameter, `n`, whose argument we will calculate the factorial for
  - A base case, such that when $n < 2$ we return `return 1`
  - A recursive case, that moves towards the base case, such that when $n \geq 2$ we `return n*fact(n-1)`

Writing a recursive function

- Our recursive function `fact` requires:
  - One `integer` parameter, `n`, whose argument we will calculate the factorial for
  - A base case, such that when $n < 2$ we return `return 1`
  - A recursive case, that moves towards the base case, such that when $n \geq 2$ we `return n*fact(n-1)`
- From these requirements, we can easily write our recursive function as:

Writing a recursive function

- Our recursive function `fact` requires:
  - One `integer` parameter, `n`, whose argument we will calculate the factorial for
  - A base case, such that when $n < 2$ we return `return 1`
  - A recursive case, that moves towards the base case, such that when $n \geq 2$ we `return n*fact(n-1)`
- From these requirements, we can easily write our recursive function as:

```
int fact(int n)
{
    if(n < 2)
        return 1;
    else
        return n*fact(n-1);
}
```

Writing a recursive function

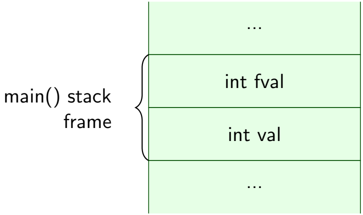
- When writing a recursive function, we must always write:
  - One or more base cases that prompt our function to return without further recursion
  - One or more recursive cases that moves us closer towards meeting the base case(s)

Overview

Overview

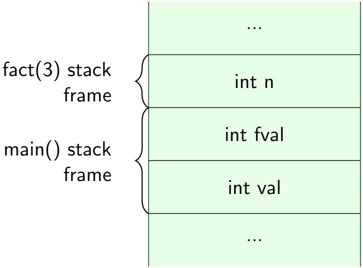
Recursive functions and the call stack: factorial

- Let's consider the state of the `call stack` as our program uses our recursive function `fact` to solve 5!
- Our program starts from `main()`, so a `stack frame` (`activation record`) for `main()` is pushed to the stack
  - Assume that main has two local variables, `int val` and `int fval` storing the value to calculate the factorial of and the return value of `fact(val)` respectively



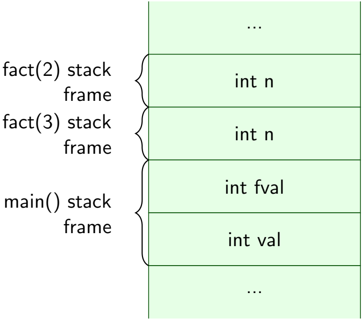
Recursive functions and the call stack: factorial

- Let's assume that our recursive function `fact` is called from main with the argument `3`
- This prompts a `stack frame` for `fact(3)` to be pushed to the `stack`
- `fact(3)` stores its argument `3` in the `local variable` `n` in its `stack frame`
- Execution has been transfered from `main()` to `fact(3)`



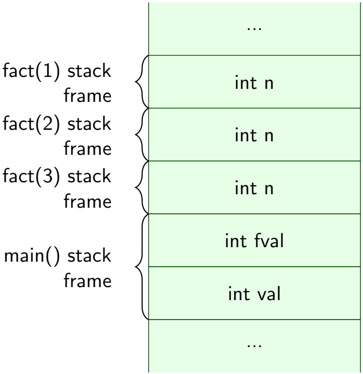
Recursive functions and the call stack: factorial

- As $n \geq 2$, we will execute the recursive case of the `fact` function, `return n*fact(2)`; `fact(2)` must be evaluated before the expression in the return statement can be evaluated
- A `stack frame` for fact(2) is thus pushed to the stack and execution is transfered to fact(2)
- `fact(2)` stores its argument `2` in the `local variable` `n` in its `stack frame`



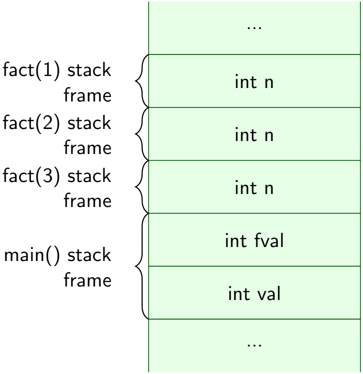
Recursive functions and the call stack: factorial

- As $n \geq 2$, we will execute the recursive case of the `fact` function, `return n*fact(1)`; `fact(1)` must be evaluated before the expression in the return statement can be evaluated
- A `stack frame` for fact(1) is thus pushed to the stack and execution is transfered to fact(1)
- `fact(1)` stores its argument `1` in the `local variable` `n` in its `stack frame`



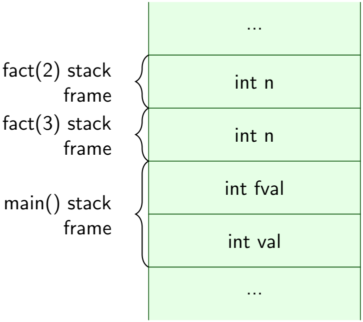
Recursive functions and the call stack: factorial

- As $n < 2$, we have finally arrived at the base case of the `fact` function, `return 1;` this statement is evaluated immediately
- `fact(1)` returns the value `1` to its caller, `fact(2)`



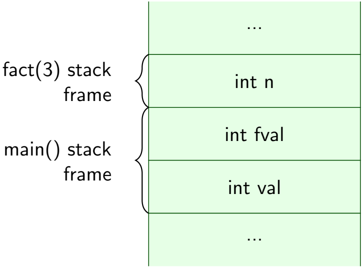
Recursive functions and the call stack: factorial

- When `fact(1)` returns the value `1`, its `stack frame` is popped from the `stack`
- Execution picks back up where things left off in `fact(2)` at the `return n*fact(1)` statement
- The return value of `fact(1)` is used in place of `fact(1)` call and `fact(2)` returns the product of `2*1` (`2`) to its caller, `fact(3)`



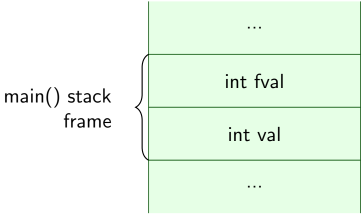
Recursive functions and the call stack: factorial

- When `fact(2)` returns the value 2, its `stack frame` is popped from the `stack`
- Execution picks back up where things left off in `fact(3)` at the `return n*fact(2)` statement
- The return value of `fact(2)` is used in place of `fact(2)` call and `fact(3)` returns the product of `3*2` (6) to its caller, `main()`



Recursive functions and the call stack: factorial

- When `fact(3)` returns the value 6, its `stack frame` is popped from the `stack` and our calculation of 3! using our recursive function is complete



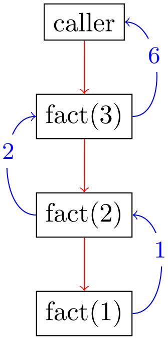
Recursive functions and the call stack: factorial

- Instead of illustrating this using a vertical stack, we will draw things using a tree structure, where each new `stack frame` is presented below the one that called it

Recursive functions and the call stack: factorial

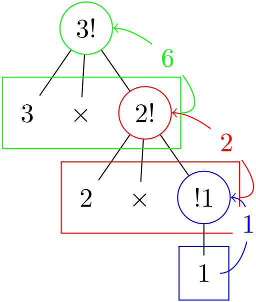
- Instead of illustrating this using a vertical stack, we will draw things using a tree structure, where each new `stack frame` is presented below the one that called it
- In our tree representation, black boxes will be used to represent `stack frames`, straight red arrows depicting function calls, and curved blue arrows with values denoting return values

Recursive functions and the call stack: factorial



Recursive functions and the call stack: factorial

- Perhaps the following diagram will help detail better what's going on and where
  - The blue, red, and green circles represent the function call to `fact(1)`, `fact(2)`, and `fact(3)` respectively
  - The blue, red, and green rectangles represent the expressions evaluated in `fact(1)`, `fact(2)`, and `fact(3)` respectively
  - The blue, red, and green curved lines detail the return value of the expressions evaluated in `fact(1)`, `fact(2)`, and `fact(3)` respectively



Overview

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call
- Let's consider the Fibonacci numbers, a sequence of numbers where each number is defined as the sum of the previous two:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3, \\ fib(n-1) + fib(n-2) & \textit{otherwise} \end{cases}$$

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call
- Let's consider the Fibonacci numbers, a sequence of numbers where each number is defined as the sum of the previous two:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3, \\ fib(n-1) + fib(n-2) & \textit{otherwise} \end{cases}$$

  - $fib(1) = 1$

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call
- Let's consider the Fibonacci numbers, a sequence of numbers where each number is defined as the sum of the previous two:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3, \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

  - $fib(1) = 1$
  - $fib(2) = 1$

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call
- Let's consider the Fibonacci numbers, a sequence of numbers where each number is defined as the sum of the previous two:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3, \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

  - $fib(1) = 1$
  - $fib(2) = 1$
  - $fib(3) = fib(2) + fib(1)$

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call
- Let's consider the Fibonacci numbers, a sequence of numbers where each number is defined as the sum of the previous two:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3, \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

  - $fib(1) = 1$
  - $fib(2) = 1$
  - $fib(3) = fib(2) + fib(1)$
  - $fib(4) = fib(3) + fib(2)$

Recursive functions and the call stack: Fibonacci

- Our recursive function `fact()` included one recursive call
- Let's consider the Fibonacci numbers, a sequence of numbers where each number is defined as the sum of the previous two:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3, \\ fib(n-1) + fib(n-2) & \textit{otherwise} \end{cases}$$

  - $fib(1) = 1$
  - $fib(2) = 1$
  - $fib(3) = fib(2) + fib(1)$
  - $fib(4) = fib(3) + fib(2)$
  - etc.

Recursive functions and the call stack: Fibonacci

- We can write the a function that calculates the value of the nth Fibonacci number recursively as:

Recursive functions and the call stack: Fibonacci

- We can write the a function that calculates the value of the *n*th Fibonacci number recursively as:

```
int fib(int n)
{
    if(n < 3)
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
```

Recursive functions and the call stack: Fibonacci

- Let's consider a call to our recursive function `fib()` that calculates the value of the 4th Fibonacci number

Recursive functions and the call stack: Fibonacci

- Let's consider a call to our recursive function `fib()` that calculates the value of the 4th Fibonacci number
- Instead of illustrating this using a vertical stack, we will draw things using a tree structure, where each new `stack frame` is presented below the one that called it

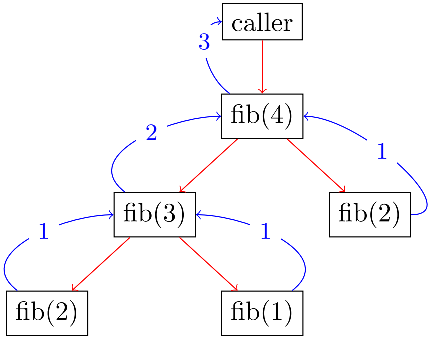
Recursive functions and the call stack: Fibonacci

- Let's consider a call to our recursive function `fib()` that calculates the value of the 4th Fibonacci number
- Instead of illustrating this using a vertical stack, we will draw things using a tree structure, where each new `stack frame` is presented below the one that called it
- In our tree representation, black boxes will be used to represent `stack frames`, straight red arrows depicting function calls, and curved blue arrows with values denoting return values

Recursive functions and the call stack: Fibonacci

- Let's consider a call to our recursive function `fib()` that calculates the value of the 4th Fibonacci number
- Instead of illustrating this using a vertical stack, we will draw things using a tree structure, where each new `stack frame` is presented below the one that called it
- In our tree representation, black boxes will be used to represent `stack frames`, straight red arrows depicting function calls, and curved blue arrows with values denoting return values
- We will assume function calls are processed from left to right; in C++ the order of such evaluation is up to the implementation (undefined)

Recursive functions and the call stack: Fibonacci



Overview

Recursion vs. iteration

- Factorial and the Fibonacci sequence are common examples of recursion

Recursion vs. iteration

- Factorial and the Fibonacci sequence are common examples of recursion
  - You can fairly easily write an iterative solution for calculating $n!$ (we already did this!) or the nth Fibonacci number (why not try at home?)

Recursion vs. iteration

- Factorial and the Fibonacci sequence are common examples of recursion
  - You can fairly easily write an iterative solution for calculating $n!$ (we already did this!) or the nth Fibonacci number (why not try at home?)
- In general, anything solved recursively has an iterative solution

Recursion vs. iteration

- Factorial and the Fibonacci sequence are common examples of recursion
  - You can fairly easily write an iterative solution for calculating $n!$ (we already did this!) or the nth Fibonacci number (why not try at home?)
- In general, anything solved recursively has an iterative solution
  - Sometimes the iterative version is more efficient, other times it is not

Recursion vs. iteration

- Factorial and the Fibonacci sequence are common examples of recursion
  - You can fairly easily write an iterative solution for calculating n! (we already did this!) or the nth Fibonacci number (why not try at home?)
- In general, anything solved recursively has an iterative solution
  - Sometimes the iterative version is more efficient, other times it is not
  - In some problems, a recursive solution maybe shorter to write and/or more elegant in nature; this may not be the case for other problems

Overview

References

- Lewis, M. C. (2015). *Introduction to the art of programming using Scala*. CRC Press.
- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.