COMP20230: Data Structures & Algorithms
Lecture 6: More Recursion

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Outline

- Recursive vs Iterative Functions
- Tail Recursion
- Turning a recursive algorithm into an iterative one
- Complexity of recursive functions

Take home message

$Running time = operation \times activations$

Tail recursion helps to turn recursive functions into iterative ones

Recursive vs. Iterative

It is often possible to write the same algorithm using recursive or iterative functions.

Algorithm Iterative implementation of factorial

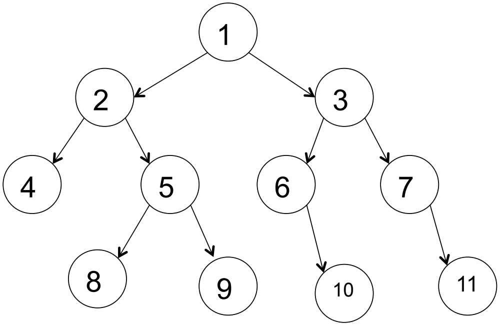
factorial_iterative(n):
Input: n a natural number
Output: the n-th factorial number

```
fact ← n
while n > 1 do
    n ← n-1
    fact ← fact * n
endwhile
return fact
```

How to choose: Recursive vs. Iterative

Naturally Recursive?

Some data structures are naturally recursive, i.e., it's much easier to write recursive algorithms for them than iterative ones.



Recursion is not always best

- Recursive functions are sometimes slower: operation calls are expensive in practice (see lab for examples)
- (Bad) recursive algorithms can generate a large number of calls
- They can sometimes be difficult to understand – elegant but illusive
- (Well written) iterative programs can be easier to follow

Tail Recursion

- A function call is said to be **tail recursive** if there is nothing to do after the function returns except return its value
- A function is non tail recursive if there is some processing done after the function returns.

Examples?

Let's look at yesterday's `factorial` as a **tail recursive** and **non-tail recursive** algorithm

Tail Recursion

Example 1: non tail recursion

This is how we implemented factorial yesterday

Algorithm *factorial_non_tail*(n)

Input: n, a natural number
Output: the nth factorial number

if $n = 1$ **then**
    return 1
else
    return $\boxed{n*}\,\mathit{factorial\_non\_tail}(n-1)$ $\quad$ # note n*rec. call
endif

We are multiplying the return value by n

Example 1: Non-tail Recursion

6

```
factorial_non_tail(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial_non_tail (2)
        endif
```

2

```
factorial_non_tail(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial_non_tail (1)
        endif
```

1

```
factorial_non_tail(1)
        if 1=1 then
                return 1
        endif
```

Tail Recursion

Example 2: Tail Recursion

Note the differences in the base call and recursion call

Algorithm *factorial_tail*(*n*, *accumulator*)

Input: n and accumulator, two natural numbers

Output: the nth factorial number

```
if n = 1 then
    return accumulator    # Note: new accumulator variable
else
    return factorial_tail(n-1, n*accumulator)    # Note n * gone
endif
```

We are using an accumulator for the total and finish at the tail

Example 2: Tail Recursion

```
factorial_tail(3,1)
        if 3=1 then
                return 1
        else
                return factorial_tail(2,3*1)
        endif
```

```
factorial_tail(2,3)
        if 2=1 then
                return 3
        else
                return factorial_tail(1,3*2)
        endif
```

```
factorial_tail(1,6)
        if 1=1 then
                return 6
        else...
```

Why use Tail Recursion

- Tail recursion is usually more efficient (although more difficult to write) than non tail recursion
- The recursive calls do not need to be added to the call stack: there is only one, the current call, in the stack
- It is possible to turn tail recursions into iterative algorithms

Tail Recursion: General Form

- `ret`: the returned type
- `param`: list of parameters
- `cond`: base case condition
- `state0`,`state1`,`state2`: statements
- `fun`: function transforming the parameters

Algorithm *generic_recursion(param)*

Input: a set of parameters, param
Output: ret, the return type

```
state0
if cond then
    state1
else
    state2
    generic_recursion(fun(para))
endif
```

Transforming: Recursive to Iterative

Algorithm *generic_recursion(param)*

Input: a set of parameters, param
Output: ret, the return type

```
state0
if cond then
    state1
else
    state2
    generic_recursion(fun(para))
endif
```

Algorithm *generic_iterative(param)*

Input: a set of parameters, param
Output: ret, the return type

```
state0
while non cond do
    state2
    para ← fun(para)
    state0
endwhile
state1
```

Example: Factorial Recursive

Algorithm *factorial_tail*$(n, accumulator)$

Input: n and accumulator, two natural numbers

Output: the nth factorial number

if $\boxed{n = 1}$ **then** # cond
    $\boxed{\textbf{return}\ accumulator}$ # state1
else
    return $\boxed{factorial\_tail(n - 1, n * accumulator)}$ # fun(para)
endif

Example: Factorial Iterative

Algorithm $factorial\_iterative(n, result)$

Input: n and result, two natural numbers

Output: the nth factorial number

while $\boxed{n > 1}$ **do** #non cond

  $\boxed{result \leftarrow n * result}$ # fun(para)

  $\boxed{n \leftarrow n - 1}$ # fun(para)

endwhile

$\boxed{\textbf{return}\ result}$ # state1

Recursion $\rightarrow$ Iterative Algorithm

- When you want to write iteratively a recursive function, the first technique is to come up with a tail recursion and then use the solution presented in previous slides
- Otherwise you need to store context of the calls (in a way, re-doing the call stack in the program)
  - use extra structures (e.g., arrays) to store the intermediary results.
  - This is an example of what is called *dynamic programming*

Example: Recursion $\rightarrow$ Dynamic Iterative Algorithm

Algorithm *factorial_dynamic*(n)

Input: n, a natural number
Output: the nth factorial number

```
array ← array of size n
array[0] ← 1
for i from 2 till n do
    array[i-1] ← array[i-2] * i
endfor
return array[n-1]
```

Recursive Complexity

Cannot apply the same mechanism used for iterative algorithms
Just counting the number of basic operations and loops will not work

Assess Two Things

1. Number of basic operations in each activation of the recursion (this is easy – same as for iterative)
2. Number of activations (this is a little more difficult)

Factorial Non-tail Recursion Complexity

Algorithm $factorial\_non\_tail(n)$

Input: n, a natural number
Output: the nth factorial number

```
if n = 1 then          # 1 operation
    return 1           # 1 operation
else
    return n * factorial_non_tail(n − 1)      # 3 operations
endif
```

Computing Complexity

5 operations per activation.
How many activations?

Counting Activiations

3 activations for $n = 3$

6

```
factorial(3)
        if 3=1 then
                return 1
        else
                return 3 * factorial(2)
        endif
```

2

```
factorial(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial(1)
        endif
```

1

```
factorial(1)
        if 1=1 then
                return 1
        endif
```

Complexity

- number of activations: $\mathcal{O}(n)$ (3 for $n = 3$)
- number of operations: 2 for base case, 4 otherwise (constant running time anyway) $\rightarrow \mathcal{O}(4) = \mathcal{O}(1)$
- Total: $\mathcal{O}(n)$, or $T(n) = 4n$, $\mathcal{O}(4n) = \mathcal{O}(n)$

Proof using Induction

General Case

$T(n) \sim 4n$

Case $n = 1$

$T(1) = 2$

Case n=2

$T(2) = 6$
$T(2) = T(1) + 4 = 6 \sim 4(2)$ (i.e. the general case)

Case n

$T(n) = T(n-1) + 4$
$T(n) \sim 4(n-1) + 4$
$T(n) \sim 4n$

Writing your own recursive algorithms

1. **Test for base cases**. Test for one or more base cases: every possible chain of recursive calls should eventually reach a base case that doesn't need recursion.
2. **Recur**. Perform one or more recursive calls. Recursive calls should make progress towards a base case.
3. **Sub-problems**. They should the same general structure as the original problem. Use concrete examples to help define.
4. **Parameters**. What you pass in helps define the problem. Sometimes the main problem will have obvious parameters but the sub-problem will be easier to define with additional parameters. You can keep the interface clean by overloading or having non-public methods to handle the messy parameters, e.g.
   `binary_search(data, target)` and
   `search(data, target, low, high)`

What is missing?

- Here we have only one sort of analysis: complexity analysis
- We have not analysed for:
  - **correctness**: the algorithm does what it claims
  - **termination**: the algorithm terminates

Conclusions

Concepts Today

- **tail recursion**: a recursion which does not process further the result of the recursive calls
- to get a tail recursion we use an **accumulator**
- turning a **tail recursion into an iterative function** is easy
- **complexity** of recursive functions requires you to estimate the number of **activations** and the **number of basic operations in each call**