Programming Fundamentals

More Recursion

João Correia Lopes

INESC TEC, FEUP

29 November 2018

GOALS

By the end of this class, the student should be able to:

- Identify some complex problems, that may otherwise be difficult to solve, that may have a simple recursive solution
- Describe how to formulate programs recursively
- Describe recursion as a form of iteration
- Implement the recursive formulation of a problem

BIBLIOGRAPHY

- Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures using Python (Chapter 4) [HTML]
- Brad Miller and David Ranum, How to Think Like a Computer Scientist: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Chapter 15) [HTML]

Tips

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

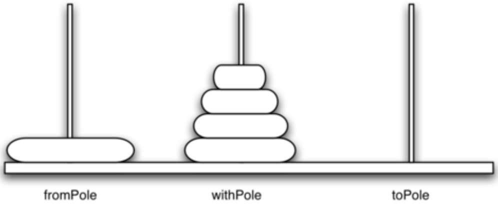
CONTENTS

CODE, TEST & PLAY

- Have a look at the code in GitHub:
  `https://github.com/fpro-admin/lectures/`
- Test before you sumit at:
  `http://fpro.fe.up.pt/test/`
- Pay a visit to the playground:
  `http://fpro.fe.up.pt/play/`

TOWER OF HANOI

- The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883 ($\Rightarrow$ Wikipedia)
- The priests were given three poles and a stack of 64 gold disks
- Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints:
  - They could only move one disk at a time, and
  - They could never place a larger disk on top of a smaller one



http://interactivepython.org/runestone/static/pythonds/Recursion/TowerofHanoi.html

TOWER OF HANOI (2)

- The number of moves required to correctly move a tower of 64 disks is

$$2^{64} - 1 = 18446744073709551615$$

- At a rate of one move per second, that is: 584 942 417 355 years!
- ⇒ [Tower of Hanoi | GeeksforGeeks]
- Pseudo-code:
  1. Move a tower of $height - 1$ to an intermediate pole, using the final pole
  2. Move the remaining disk to the final pole
  3. Move the tower of $height - 1$ from the intermediate pole to the final pole using the original pole

⇒ https://github.com/fpro-admin/lectures/blob/master/18/hanoi.py

TOWER OF HANOI (2)

- The number of moves required to correctly move a tower of 64 disks is

$$2^{64} - 1 = 18446744073709551615$$

- At a rate of one move per second, that is: 584 942 417 355 years!
- ⇒ Tower of Hanoi | GeeksforGeeks
- Pseudo-code:
  1. Move a tower of $height - 1$ to an intermediate pole, using the final pole
  2. Move the remaining disk to the final pole
  3. Move the tower of $height - 1$ from the intermediate pole to the final pole using the original pole

⇒ `https://github.com/fpro-admin/lectures/blob/master/18/hanoi.py`

ITERATION VS. RECURSION

- Recursion and iteration perform the same kinds of tasks:
  - Solve a complicated task one piece at a time, and combine the results
- Emphasis of iteration:
  - keep repeating until a task is finished
  - e.g. loop counter reaches limit, list reaches the end, ...
- Emphasis of recursion:
  - Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results
  - e.g. recursive factorial function

Sum of a List of Numbers

- We will begin our investigation with a simple problem that you already know how to solve without using recursion
- Suppose that you want to calculate the sum of a list of numbers such as:

$$[1, 3, 5, 7, 9]$$

SUM OF A LIST OF NUMBERS ITERATIVE

- The function uses an accumulator variable (`the_sum`) to compute a running total of all the numbers in the list by starting with `0` and adding each number in the list

⇒ https://github.com/fpro-admin/lectures/blob/master/18/listsum_iter.py

SUM OF A LIST OF NUMBERS RECURSIVE

- The sum of a list of length 1 is **trivial**; it is just the number in the list
- The series of (recursive) calls may be seen as a **series of simplifications**

$$(1 + (3 + (5 + (7 + 9))))$$

- Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller

⇒ `https://github.com/fpro-admin/lectures/blob/master/18/listsum_rec.py`

FACTORIAL RECURSIVE

```
def fact_rec(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n * fact_rec(n-1)
```

- O(n)
- Look at **tail recursion** (⇒ Neopythonic)

FACTORIAL RECURSIVE

```python
    def fact_rec(n):
        """ assume n >= 0 """
        if n <= 1:
            return 1
        else:
            return n * fact_rec(n-1)
```

- O(n)
- Look at **tail recursion** (⇒ Neopythonic)

FACTORIAL ITERATIVE

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod = i * prod
    return prod
```

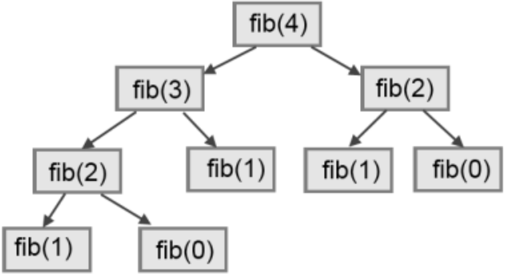
- O(n)
- Is it easier to read?
- Is it faster?

FACTORIAL ITERATIVE

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod = i * prod
    return prod
```

- O(n)
- Is it easier to read?
- Is it faster?

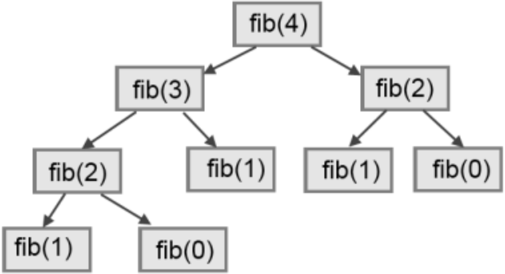
FIBONACCI RECURSIVE



- $O(2^n)$
- It is a binary tree of height n: for $n = 4$ we have $2^n - 1 = 15$ nodes
- $\Rightarrow$ StackExchange

$\Rightarrow$ https://github.com/fpro-admin/lectures/blob/master/18/fib_rec.py

FIBONACCI RECURSIVE



- $O(2^n)$
- It is a binary tree of height n: for $n = 4$ we have $2^n - 1 = 15$ nodes
- $\Rightarrow$ StackExchange

$\Rightarrow$ `https://github.com/fpro-admin/lectures/blob/master/18/fib_rec.py`

FIBONACCI EFFICIENT

- Calling `fib(34)` results in 11405773 recursive calls to the procedure
- Calling `fib_efficient(34)` results in 65 recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient (*memoisation*)

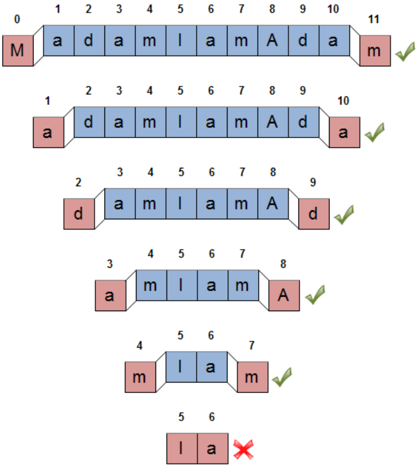
⇒ https://github.com/fpro-admin/lectures/blob/master/18/fib_efficient.py

FIBONACCI ITERATIVE

- $O(n)$ (one `for` cycle)

⇒ https://github.com/fpro-admin/lectures/blob/master/18/fib_iter.py

IS A PALINDROME RECURSIVE



⇒ https://github.com/fpro-admin/lectures/blob/master/18/is_palindrome_rec.py

IS A PALINDROME ITERATIVE

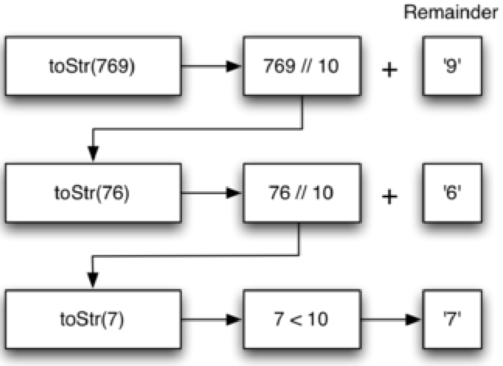
- $O(n)$ (complexity of `join` method)

⇒ https://github.com/fpro-admin/lectures/blob/master/18/is_palindrome_iter.py

Converting an Integer to a String in Any Base

- Suppose you want to convert an integer to a string in some base between binary and hexadecimal
- While there are many approaches one can take to solve this problem, the recursive formulation of the problem is very elegant
  1. Reduce the original number to a series of single-digit numbers
  2. Convert the single digit-number to a string using a lookup
  3. Concatenate the single-digit strings together to form the final result

Converting an Integer to Base 10



⇒ https://github.com/fpro-admin/lectures/blob/master/18/to_base.py

RECURSION VS. ITERATION

- **Advantages of Python Recursion**
  - Recursive functions make the code look clean and elegant
  - Very flexible in data structure like *tree traversals*, *stacks*, *queues*, *linked list*
  - Big and complex iterative solutions are easy and simple with Python recursion
  - Sequence generation is easier with recursion than using some nested iteration
  - Algorithms can be defined recursively making it much easier to visualize and prove
- **Disadvantages of Python Recursion**
  - Sometimes the logic behind recursion is hard to follow
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time[1]
  - More difficult to trace and debug
  - Recursive functions often throw a *Stack Overflow Exception* when processing or operations are too large

[1] For every recursive call separate memory is allocated for the variables

Summary about Recursion

- All recursive algorithms must have a base case
- A recursive algorithm must change its state and make progress toward the base case
- A recursive algorithm must call itself (recursively)
- Recursion can take the place of iteration in some cases
- Recursive algorithms often map very naturally to a formal expression of the problem you are trying to solve
- Recursion is not always the answer: sometimes a recursive solution may be more computationally expensive than an alternative algorithm.

Exercises

- Moodle activity at: LE18: More recursion