COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 5: Feb 05

Lecturer: Dr. Andrew Hines *Scribes: Scarlet Grant, Fabio Magarelli*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

5.1 Outline

List of recommended books: It is recommended to read a book on data structures. In the slides some examples. You might prefer the one for Python:

- Search pdf on Google for: Data Structures and Algorithms in Python - Authors: Goodrich, Tamassia and Goldwasser

This lecture covered an introduction to Recursion with 3 key points:

- Recursion
- Base case
- Call stack

The next lecture we will cover: Recursive and Iterative Functions, Tail Recursion, and Complexity of recursive functions.

Take home message of this lecture: Recursion is a method to divide a problem in similar sub-problems. Simplify repetition using a function calling itself.

Recursion explained in pictures:

- 54321 chocolate jingle (simplicity)
- Recursive Droste tins (repetition, calling itself)
- Pringles stacks (put on the top, take off the top)
- Cricket Back stops (throw recursionError)

5.2 Recursion

Lecture opened with a Jingle of 54321 to denote the simplicity of recursion. We moved on to the subject with a picture of a 'Droste tin case' showing the tin in an image which shows the tin in an image etc.

Definition recursion: *a method to divide a problem in similar sub-problems. Recursion simplifies repetition using a function calling itself. Recursion is a way of decomposing problems into smaller, simpler sub-tasks that are similar to the original.*

Technique: The whole problem is solved by the sum of all the sub-problems solutions. The fundamental technique is repeated over and over again.

2 conditions are required:

- It has a Base case (a case simple enough to solve without recursion)
- It has a Stop condition (to end the recursion instead of infinite looping)

Classic recursion algorithms also contain:

- if-else;
- reference to itself; it is calling itself but it is changing it's input.

5.3 Base case

The base case is a case that is simple enough so that does not require recursion.

An example of recursion is the factorial. Other examples where recursion is used are:

- Certain design patterns, often occurring in nature
- The Fibonacci sequence

Algorithm 1: Algorithm factorial(n)

```
1 if n = 1 then
2 return 1
3 else
4 return n * factorial(n-1)
5 endif
```

Side note on complexity: Increased complexity of algorithm could lead to decreased complexity of the data structure or vice versa. The use of complex data structures such as dictionaries could make our algorithm easier to read by our selves and other developers in our team improving the maintainability of our code.

Keep code *simple*, *readable* and *consistent*. It you are not able to follow it yourself when you come back to it, then don't expect others will either.

5.3.1 Stop case / Stop condition

The recursive operation calls itself changing the input argument(s). We want a stopping condition (backstop) that get's tested to check if the function is going to stop. If there is no base case, a "Stack overflow" can occur (See 5.4). In the code previously, the 'base case' is 1 because if we call 'factorial(1)' it will return 1 thus the recursion end.

Backstop metaphor: The term backstop is used in cricket for the player that takes the position in the field as "backstop". All (s)he does is catching the balls before they eventually pass the outer lines of the play field.

If we don't have a base case, the recursive function will continue forever or more plausible, will throw a stack overflow error. Python has a backstop which will throw a RecursionError. Python backstop has a default value which identify the maximum number of recursion allowed. This limit can also be changed by the developer to allow more recursions thus allocating more space in memory to the program.

Algorithm 2: Recursion limit in Python

```
1 import sys
2 return the limit number of allowed recursion before throwing a recursionError.
3 sys.getrecursionlimit()
4 Out: 1000
```

5.4 Call Stack

A stack is a data structure:

- You can only put things on the top of the stack
- You can only get things off the top of the stack

It can be compared to a pack of "Pringles": you can only eat Pringles from the top and you can only add Pringles from the top.

To see how a Stack works, let's try running 'factorial(3)' from the previous code:

Algorithm 3: Output of factorial(3) when we run algorithm 1

```
1 Stack: (from bottom to top)
2 - factorial (1) //base case
3 - factorial (2)
4 - factorial (3)
5 Stack:
6 - factorial (2)
7 - factorial (3)
8 Stack:
9 - factorial (3)
10 ... empty stack
```

Stack overflow: A stack overflow occurs if there is no more room in call stack. In this case, we say that the Stack has reached the boundary in the room, and there is no more space available. If my stack has less space than needed, I get a stack overflow. To avoid stack overflow or recursionError exception (in Python), we have to define a base case from the moment we write our pseudocode.

5.4.1 Why Recursion Works

To explain recursion in a different way: In a recursive algorithm, the computer 'remembers' every previous state of the problem. This information is 'held' by the computer on the 'activation stack' (i.e., inside of each functions workspace). Every function has its own workspace PER CALL of the function. [Utah, 2008] See more on the site of University of Utah.

5.5 Conventions and general considerations

Question: *Is it better to have code that is more 'maintainable' or more 'efficient'?*

Even if recursive operations are not efficient, they can be very clear to code and to debug. If we increase the complexity of the data structure, we can decrease the complexity of the algorithm and vice versa.

Examples:

Dictionaries are complex data structure that makes algorithms easier to read and to write but you could actually achieve the same results without using them.

File structure is a kind of data structure (tree structure) which is a good example on where it might be useful to use a recursive operation.

5.6 References

1. Cs.utah.edu. (2008). H. James de St. Germain. Programming - Recursion . [online] Available at: https://www.cs.utah.edu/ germain/PPS/Topics/recursion.html [Accessed 7 Feb. 2019].