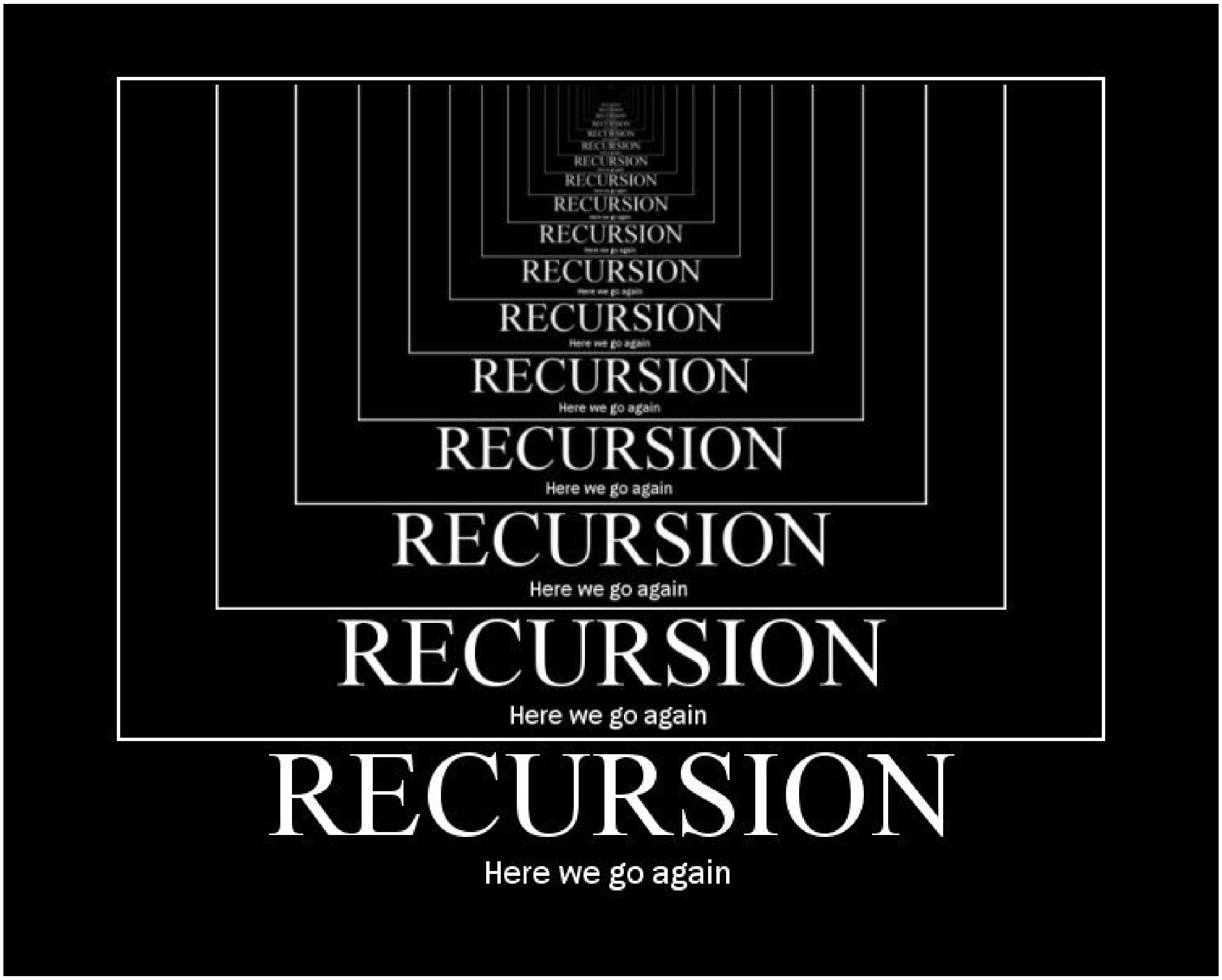
COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 5: Recursion

Lecturer: Dr. Andrew Hines *Scribes: Raphael Hetherington, Stephen Keenan*



Contents

5.1 Learning Outcomes

- Understand what recursion is
- Understand why it's worth knowing
- Understand how it works, and discuss other terminology such as:
  - Base cases
  - Call stacks

5.2 Recursion Explained

”*Recursion (n.)* A computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first.”[1]

- As stated in the dictionary definition above, recursion is the process of a function calling itself multiple times to produce an output.
- Recursive solutions to computational problems are often more simple than iterative ones, but this simplicity is counterbalanced by the demands on memory of recursive programs, the opportunity for miscalculation, and, not least, the threat thinkng recursively presents to the sanity of the programmer.
- Fractals in nature represent a type of recursive activity; the trunk of a tree extending to the branches, smaller branches and twigs is analogous to the ’race to the bottom’ that occurs in recursive algorithms.
- Another analogous recursive situation is one mirror placed opposite another; the one will recursively reflect the other and vice versa.

5.2.1 Thinking recursively

A problem can be solved recursively if it is possible to break it down into equivalent smaller problems. As we shall see, a typical use-case for recursion is calculating the factorial of a number. In this case we can say that:

$$n! = 1 \times 2 \times \ldots (n-1) \times n \tag{5.1}$$

Which is the same as

$$n! = (n-1) \times (n-2) \ldots \tag{5.2}$$

Which is

$$n! = (n-1)! \text{ and } 0! \ = 1 \tag{5.3}$$

Thus we can see the problem *n!* consists of a repetition of consecutively smaller iterations of the same problem. This means that we can solve it recursively.

5.2.2 Writing recursive algorithms

Writing recursive algorithms can be dangerous for those who like to visualise their mathematical operations. The figure of the call stack provided below is useful for an initial understanding of how a recusive function operates, but for more complicated examples the thinking is best left to the computer. Remember the basic point: recursion is a repeating process of the same, gradually diminishing, problem. If you can describe this diminishing problem as well as provide a way for the process to end (in the form of a base case, also discussed below), there is no need for you to dive Boba Fett-like into the sarlacc’s pit of recursive calls.

[1] Merriam Webster - https://www.merriam-webster.com/dictionary/recursion

5.3 Properties of Recursion

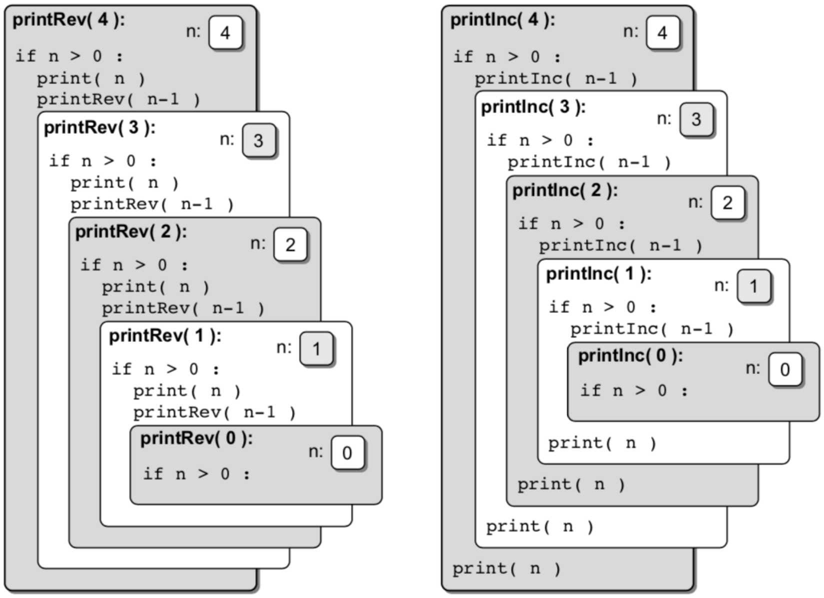
A recursive contains set properties that must be present for a recursive solution to be correct. The following are the three properties:

1. **A Recursive Case**
   For a problem to be subdivided, it must contain a data set that can be divided into small sets or a smaller term. This subdivision is handled by the recursive case when the function calls itself.

2. **A Base Case**
   Often referred to as the termination case, it represents the smallest subdivision of the problem. It signals the end of the recursive calls. The base case prevents the function from calling itself an infinite number of times.

3. **A Recursion Must Make Progress Toward the Base Case**
   The final property of a recursive solution is must make a progress towards the base case to the recursive will never stop resulting in an infinite virtual loop.

The figure below illustrates the properties in action:



- The recursive case is each subdivision of the original function call: printRev().

- The base case is illustrated through the conditional statement, where n ¿ 0. When n is less than 0, the conditional statement is not satisfied and the base case is reached.

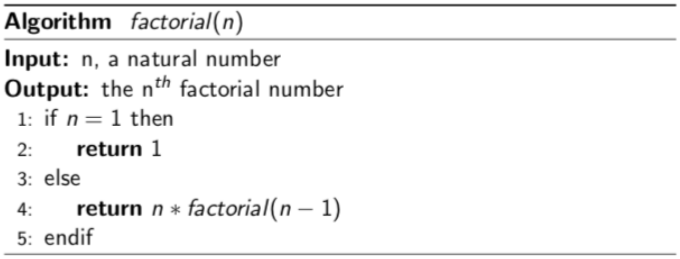
- The progress towards the base case is depicted through each function call. The systematic subdivision of the dataset is evidence of progress towards the base case: printRev(n – 1).

5.3.1 A detailed example

The above example attempts to explain recursion in its mathematical construct. The below example is one that the class would be more familiar with – a common computer science example.

In computing the factorial of a number, a recursive algorithm can be used. Below is an example of such an algorithm – in pseudocode. We can see that there is a function called ‘Factorial (n)’. It also contains all of the properties listed above:

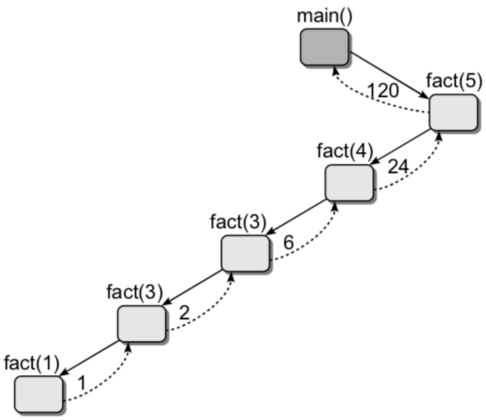
1. Recursive Case: We can see that the function is calling itself in the else condition.
2. Base Case: We can see that the function has a condition checking that the value for n is greater than 1.
3. Progress: We can see that the function is calling itself while decrementing the value of 1 every time. Indicating that there is progress present.



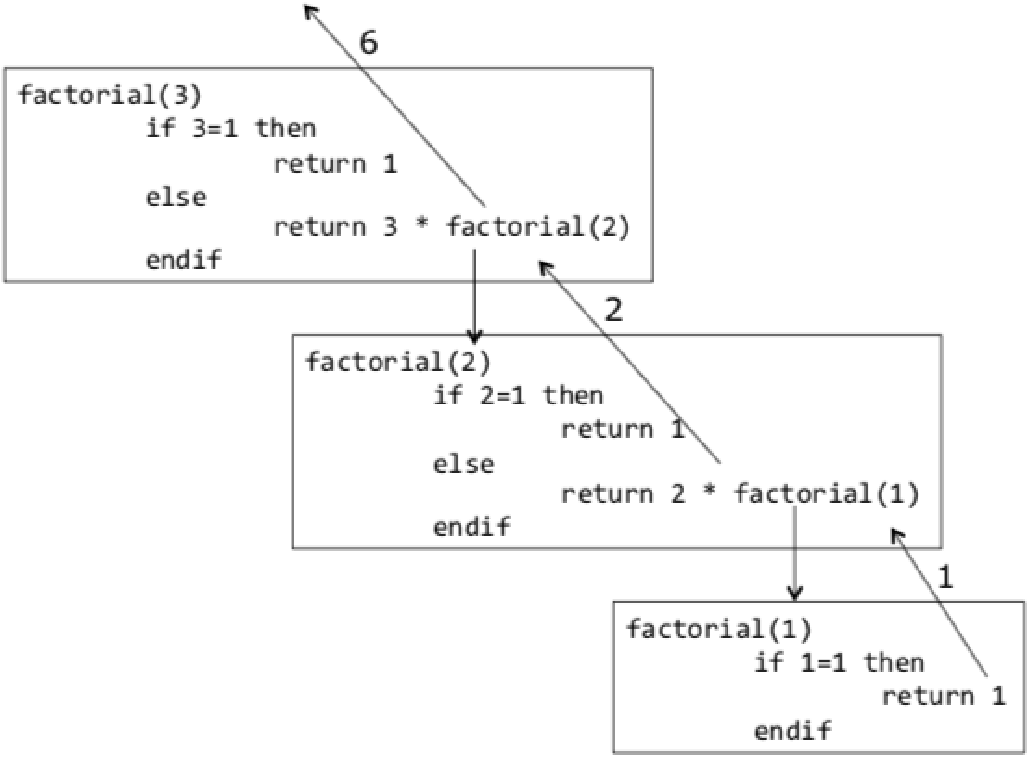
```
Algorithm  factorial(n)
Input: n, a natural number
Output: the n^th factorial number
if $n = 1$ then
    return 1
else
    return $n * factorial(n - 1)$
endif
```

Below is another figure that shows how the above algorithm works. This is often called a recursive call tree. A Recursive call tree is used to develop and evaluate a recursive algorithm. The solid arrowed lines indicate the function call:

$$main() \rightarrow fact(5) \rightarrow fact(4) \ldots etc. \tag{5.4}$$



Finally, the below illustration combines the two above diagrams. Each box represents a function call. The last box is the base case with a return value of 1. This value is then used by the calling function and the sum calculated – this process is repeated until the original function is reached.



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

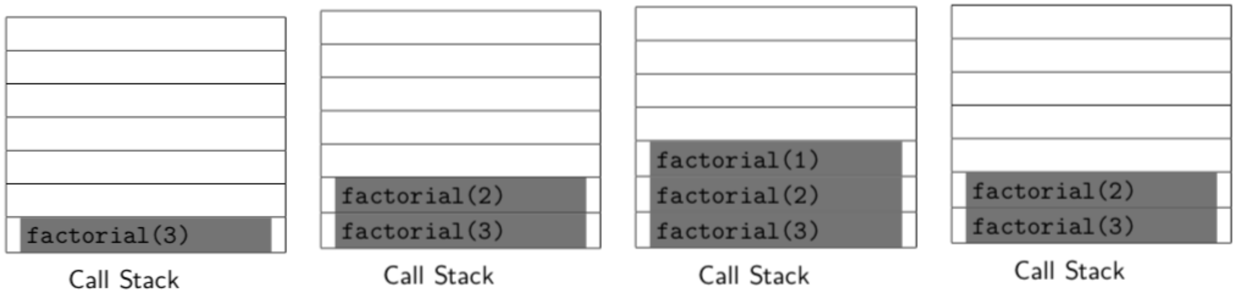
```
factorial(2)
        if 2=1 then
                return 1
        else
                return 2 * factorial(1)
        endif
```

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

5.3.2 Recursion in Computation: The Stack

In order for the factorial example to be executed, it must be allocated a stack. This allocation is done so that the program running has an available data structure on which to use to run its operation. This stack stores information about active subroutines - or in the above example subdivisions - of the computer program. The below example illustrates how the recursive call of a function to calculate factorial impacts the stack. Each recursion results in an extra call being added to the stack. This will continue until the base case is reached – in this example '1'.

Bonus
This can also be called activation record – and it is automatically created in order to maintain information related to the function. One piece of information that is automatically created and saved is the return address. This is the location of the next instruction to be executed when the function terminates.



The stack highlights the need for a base case. As a stack is assigned to a program, it is only of finite memory. Therefore, if the size of the stack is 'n', and the recursion occurs n + 1 times, we have a memory allocation issue. This is known as a stack overflow. Some languages have built-in safety procedures (Python has a RecursionError exception).

One thing to note about a stack is that only the top call can be removed before attempting to remove the one below it. This is the same as trying to eat any pringle, in a tube of pringles, that isn't the top pringle.

5.4 Conclusion

Recursion can be a very neat way to construct your code. It is concise and, once you get to grips with the concept, are easy to identify. However, the big issues with recursion is the order of magnitude of computing power that it takes. Although there are methods to reduce this (see next chapter), recursion's rate of power needed for each additional input often leads itself to be side-lined in favour of other, less computation intensive, solutions.