# Chapter 4 : Recursion

Aviral Janveja

## 1 Introduction

In computer science, recursion is an approach to solving computational problems, where the solution depends on solutions to smaller instances of the same problem. Recursive programs implement this by using functions that call themselves from within their own code. This approach can be applied to many types of problems. Let us look at a simple example :

```
def factorial(n):
    if n == 0: # Base case
        return 1
    else:
        return n * factorial(n - 1) # Recursive step

# Example usage:
result = factorial(5)
print("Factorial of 5 is :", result)
```

In the above example, the function calculates the factorial of a non-negative integer. It does so by recursively calling itself with a smaller argument $(n - 1)$ until it reaches the **base case** $(n == 0)$, at which point it returns 1. Then, as the recursion unwinds, it multiplies the return value of each recursive function call by the current value of $n$.

So, when you call factorial (5), it would call 5 * factorial (4), then 4 * factorial (3), then 3 * factorial (2), then 2 * factorial (1), and finally it reaches the base case 1 * factorial (0), at which point, recursion stops and returns the value 1. Then, the recursive chain of function calls unwind and the final result is computed as 1 * 1 * 2 * 3 * 4 * 5, hence returning 120. Checkout the above factorial example here : recursion.py. Further you can use pythontutor.com to visualize the recursion process, step by step.

### 1.1 Base Case

In recursion, the **base case** is the condition that determines when the recursive calls should stop. It is the termination condition that prevents the function from calling itself indefinitely, thereby ensuring that the recursion eventually ends.

In the above example of the factorial function, the base case is when $n$ is equal to 0. When $n$ becomes 0, the function returns 1 without making any further recursive calls.

## 2 Towers of Hanoi

The Towers of Hanoi is a mathematical puzzle consisting of three rods and a number of disks of different diameters which can slide onto any of these rods. The puzzle begins with the disks stacked on the left most tower in order from largest at the bottom to smallest at the top, thus approximating a conical shape. The puzzle can be tried out here to develop a better understanding of it.

The objective of the puzzle is to move the entire stack from left-most tower to right-most tower using the middle tower as an aid. This has to be achieved while obeying the following rules :

- No disk is placed on top of a smaller disk.
- Only one disk is moved at a time.

Let us say there are $n$ disks and the three towers are labeled as $start$, $goal$ and $extra$. We then start solving the above puzzle with $n = 1$. This is easily solved by moving the single disk from $start$ to $goal$.

Next, we solve for $n = 2$. Here, to get the bigger disk to $goal$, we first need to move the smaller disk above it. So we move the smaller disk to $extra$, the larger disk is now free to move to $goal$ and the smaller disk can now be moved to $goal$ as well, hence solving the puzzle for two disks.

Having solved for two disks already, we can now solve for three disks using the same approach. First, we move the top two disks to $extra$ using the same process as above, the third disk is now free to move from $start$ to $goal$ and then we can again move the two disks from $extra$ to $goal$. The same process can be applied to solve for $n = 4$, $n = 5$ and so on.

This is the thought process behind the recursive solution. Given $n$ disks, we think about moving $n - 1$ disks first, recursively until we reach the trivial base cases, as we have already solved them before. The following recursive procedure can be employed to find the optimal solution, while respecting the puzzle rules :

1. Move $n - 1$ disks recursively from $start$ to $extra$, one disk at a time.
2. Then move the $n^{th}$ disk from $start$ to $goal$.
3. Finally, move the $n-1$ disks recursively from $extra$ to $goal$, one disk at a time.

The above algorithm is coded here : tower_of_hanoi.py. Towers of Hanoi is one of the most natural examples for learning about recursion as the recursive solution is quite intuitive and elegant compared to the iterative approach in this case.

> **Note** : Interestingly, the origins of the Towers of Hanoi problem is tied to a legend connected with the Kashi Vishwanath temple in Varanasi, Bharat. According to legend priests at a temple in Varanasi have been carrying out the movement of the "Sacred Tower of Brahma", consisting of sixty-four golden disks, according to the same rules as in the game, and that the completion of the tower would lead to the end of the world.

# 3 Fibonacci Numbers

The Fibonacci numbers were first described in Indian mathematics as early as 200 BC in work by **Maharishi Pingala** on enumerating possible patterns of Sanskrit poetry. They are named after the Italian mathematician Leonardo of Pisa, also known as Fibonacci, who introduced the sequence to Western European mathematics in 1202. We therefore refer to them as Pingala numbers going forward.

The Pingala numbers are a sequence of numbers in which each number is the sum of the two preceding ones. Starting from 0 and 1, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 and so on. Formally, the Pingala sequence is defined as follows :

$$P(0) = 0$$
$$P(1) = 1$$
$$P(n) = P(n-1) + P(n-2) \text{ for } n \geq 2$$

You can look at the corresponding coded program here : pingala.py

It is worth mentioning that while recursion is an elegant solution for many problems, it can be inefficient for certain cases due to the overhead of function calls. For the Pingala sequence, the above implementation has an exponential run-time complexity, which means it's not very efficient for large values of n. In practice, for better performance, you would typically use an iterative approach to optimize the calculation of Pingala numbers.

# 4 Final Note

In summary, recursion is a powerful tool in the programmer's toolbox, but it should be used judiciously. Understanding when recursion is appropriate and when it may lead to inefficiencies is crucial for writing efficient and maintainable code.

In some cases, the iterative approach may be simpler, more readable and more efficient than the recursive approach. Iteration often avoids the overhead of function calls and can be more straightforward for certain types of problems.

In many cases, a balance between recursion and iteration may be necessary to achieve optimal performance and clarity in code.