DAILY DSA | DAY-21 | Recursion – Part 1|
-GOPALKRISHNA A

Just like many great ideas in history, recursion was discovered by accident. A mathematician named Edouard Lucas stumbled upon the concepts in the late 1800s while trying to solve a problem related to Fibonacci numbers. Recursion is now a fundamental technique applied by software engineers for solving complex problems.

Recursion is a subset of mathematics & computer science that deals with the idea of self-reference. Its technique allows the computer to break down a task into smaller pieces until it's easy to solve.

Analogy:
People often sort stacks of documents using a recursive method. For example, imagine you are sorting 100 documents with names on them. First, by placing documents into piles by the first letter, then sort each pile.

Recursion:
A recursive algorithm is one that can be defined in terms of itself, usually by applying the algorithm to smaller portions of the input and then combining those results together to obtain the solution for the original input.

When to use recursion?
We can recognize a recursive problem by first assuming that we have an algorithm **A** for solving the problem, and figuring out whether the solution on your original input **x** can be obtained by combining the solutions **A(x_1), A(x_2),..**. where **x_1, x_2, ...** are sub-parts of your original input **x**. This is known as a **divide-and-conquer** algorithm, as we first **divide** the problem into multiple smaller parts, delegate the sub-problems to the recursive algorithm, and **combine** the solutions of the sub-problems into the final solution for the original input.

Example: Determine if a string is palindrome

Given a string (e.g. "abcba"), return **True** if the string is a palindrome, and **False** if not. We can break down this problem by starting from both ends of the string and checking if the characters at either end of the string (e.g. "a" and "a") are equal. if they are, then it suffices to check if the remaining substring (e.g. "bcb") is also a palindrome, which we can do with a recursive call.

```
IsPalindrome("abcba") = IsEqual("a", "a") AND IsPalindrome("bcb")
```

We can continue breaking down the remaining IsPalindrome call, as follows:

```
IsPalindrome("bcb") = IsEqual("b", "b") AND IsPalindrome("c")
```

Putting together what we've come up with so far, we have:

```
IsPalindrome("abcba") = IsEqual("a", "a") AND (IsEqual("b", "b") AND IsPalindrome("c"))
```

Now the only **IsPalindrome** call remaining on the right-hand side of our expression is **IsPalindrome**("c"), for which we can trivially return **True** since we know that any single character is a palindrome. When we have reduced our logic down to the simplest possible input like this, we have reached the **base case**

Formulating a recursive algorithm

As we now know to identify a recursive problem, we have to design the algorithm for solving the recursive problem.
Typically, a recursive algorithm consists of two components:

Base case: This is the simplest possible input for the problem - A case where we can no longer break down the problem into simpler sub-problems. The base case is usually some form of empty input, such as an empty list, or 0. Sometimes base cases need multiple inputs

In the palindrome example above, we actually need two base cases:
1) **IsPalindrome("")=True** (for when the original string has an even number of characters)
2) **IsPalindrome("a")=True**, which actually applies for any input that consists only of a single character (for when the original string has an odd number of characters)

Recursive step: This is the formula by which we decompose the problem in terms of smaller sub-problems and recursively apply the algorithm to the sub-problems. Usually, this involves breaking down the inputs in some way (e.g., removing a character from a substring, removing an element from a list) and re-combining the outputs from the recursive calls in some way (e.g., adding, appending to a list)

In the palindrome example above, we can decompose the problem in terms of the first character, last character, and substring in between the first and last characters. Suppose that the input string has length **n**, the recursive step boils down to

```
IsPalindrome(input)=IsEqual(input[0], input[n-1]) AND IsPalindrome(input[1:n-1])
```

Putting this together with the best cases, our code looks something like this:

```python
def is_palindrome(input_str: str):
  n = len(input_str)
  # Base cases
  if not input_str or n == 1:
    return True
  # Recursive step
  return (input_str[0] == input_str[n-1]) & is_palindrome(input_str[1:n-1])
```