What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Need of Recursion

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Direct recursive functions

These functions are the same recursive functions that you have been learning till now. When a function calls the same function again, you call it a direct function. Factorial is an example of direct recursive as shown below:

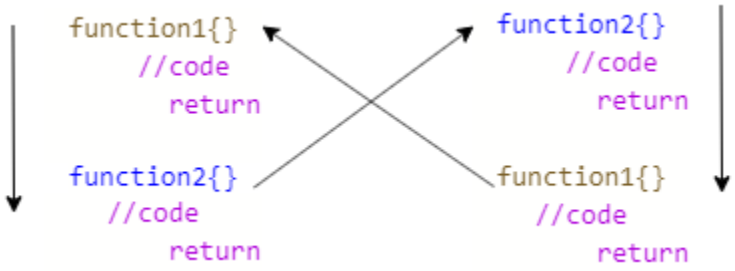
```
In [11]: def factorial(n):
          if n < 0 or n == 1:

          # The function terminates here
            return 1
          else:

                 value = n*factorial(n-1)
          return value
```

```
In [12]: # Returns the factorial of 5
         factorial(5)
```

```
Out[12]: 120
```

Indirect recursive functions

In Indirect recursive functions, two functions mutually call each other wherein both the functions must have a base case. Let's understand using a flowchart.



Here, function1 calls function2 which in turn calls function2 which calls the first function. This can be explained using an example. Suppose, you have a list of 10 numbers. For every odd number in the list, you add 1 to the number and for every even number in the list, you subtract 1. If the number is 1, output should be 2 and similarly, if the number is 2, output must be 1. You can employ indirect recursive functions to solve such problems like shown here:

In [19]:
```
def odd(n):
 if n <= 10:

    return n+1
 n = n+1
 even(n)
 return
def even(n):
 if n <= 10:
    return n-1
 n = n+1
 odd(n)
 return

# Call the function 'odd'
odd(1)
```

Out[19]: 2

In [18]:
```
# Call the function 'even'
even(2)
```

Out[18]: 1

Algorithm: Steps

The algorithmic steps for implementing recursion in a function are as follows:

Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

Tower of Hanoi: Complete Step-by-Step

The Objective of the Tower of Hanoi Problem

The objective or goal of this problem is to transfer all the 'n' discs from source pole to the destination pole in such a way that we get the same arrangement of discs as before. But this goal must be achieved by sticking to the rules.

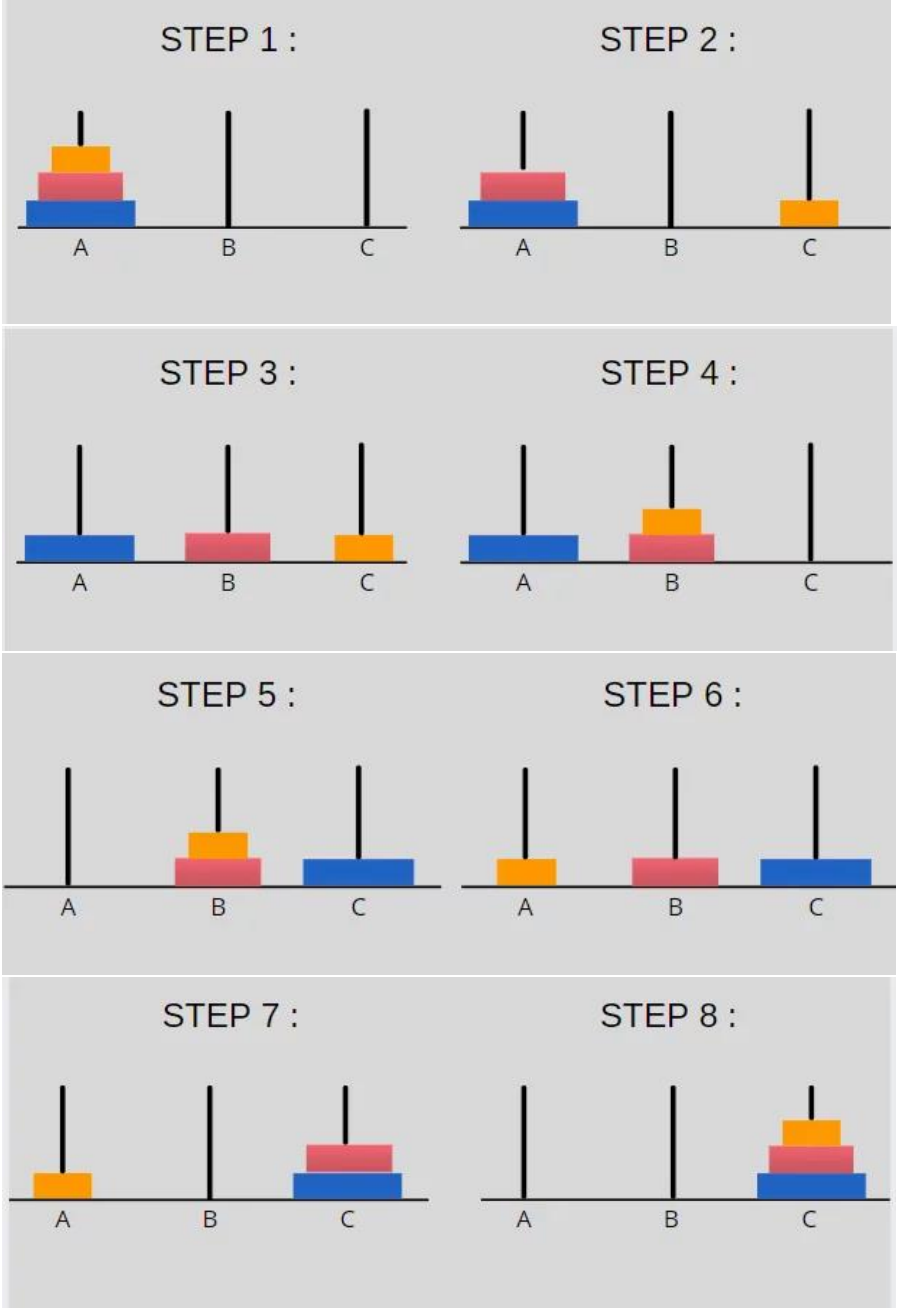
Rules and Constraints

The constraints that must be satisfied while solving the problem are –

1. Only one disc can be moved at a time.
2. Only the top-most disc can be removed
3. The larger disc cannot be placed on top of the smaller disc.

Visual Representation of the Tower of Hanoi problem

The following picture shows the step-wise solution for a tower of Hanoi with 3 poles (source, intermediate, destination) and 3 discs. The goal is to move all the 3 discs from pole A to pole C.



As we can see from the above solution, the number of moves needed for 3 discs = 8. So, a generalized formula for a total number of moves we need is:

Total number of moves = $n^2 - 1$

Where 'n' is the total no. of discs.

Real World Applications

While the Tower of Hanoi's past and present mainly involve recreational math, its future involves major real world applications. The Tower of Hanoi game can be used to assess the extent of various brain injuries and it also acts as an aid to rebuild neural pathways in the brain and to forge new connections in

the prefrontal lobe. Attempting to solve the Tower of Hanoi exercises parts of the brain that help to manage time, present a business plan or make complex arguments. Even without actually solving the puzzle, anyone who attempts to solve the Tower of Hanoi can benefit (entertainment).