Recursion

Recursion is a method of problem-solving where you solve smaller instances of the same problem through recursive algorithms.

Recursive algorithms are functions that call themselves from within their own.

**Example:**

To create a Recursive algorithm in python, create a function that calls itself.

The code below is a recursive solution to finding the factorial of a number, n. It defines a function called ‘factorial’ and calls itself ‘(factorial(n - 1))’ on the last line.

The factorial of a number is the product of all positive integers less than or equal to the number. For example, the factorial of 5 is 5 × 4 × 3 × 2 × 1 = 120.

def factorial(n):

if n == 0: #Base Case

return 1

else:

return n \* (factorial(n - 1)) #Function Calling Itself

# How Recursion Works

1. During the initial iteration of the recursive algorithm, the code inside it changes the input and passes in a new, different input for the next time it calls itself.
2. To stop the recursive algorithm from calling itself forever, there must be a base case. (A base case is a condition that ends a recursive algorithm to stop it from continuing forever.)
3. Each time the recursive algorithm calls itself, it gets closer to the base case.
4. Eventually, the base case condition is satisfied, the problem is then solved and the recursive algorithm stops calling itself.

# Three Laws of Recursion:

1. A recursive algorithm must have a base case.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself recursively

# Fibonacci sequence

The Fibonacci sequence is asequence in which each number is the sum of the two preceding ones. Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

def fibonacci (n):

if n <= 1:

return n

else:

return fibonacci (n - 1) + fibonacci (n - 2)

# When to Use Recursion

Any algorithm you can write recursively, you can also write iteratively (ie using loops)

**Recursion VS Iteration**

| **Recursion** | **Iteration** |
| --- | --- |
| Terminates when the base case becomes true. | Terminates when the condition becomes false. |
| Used with functions. | Used with loops. |
| Every recursive call needs extra space in the stack memory. | Every iteration does not require any extra space. |
| Smaller code size. | Larger code size. |

# 

# Types of Recursion:

1. **Direct recursion:** When a recursive algorithm iss called within itself directly it is called direct recursion. This can be further categorised into four types:
   * Tail recursion, (Calls itself at the end )
   * Head recursion, (Calls itself at the start)
   * Tree recursion and (calls itself more than once)
   * Nested recursion (a recursive algorithm that takes itself as a parameter)
2. **Indirect recursion:** Indirect recursion occurs when a function calls another function that eventually calls the original function and it forms a cycle.

# Advantage and Disadvantages of using Recursive Algorithms

The main advantage of recursion is how elegant it is. That means it takes you less code to solve as compared to an iterative solution.

A disadvantage of recursive algorithms is that they often take up more memory because they have to hold data on Python’s internal stack. Hence all recursive approaches can be rewritten iteratively using a stack. Beware of cases where the recursion level goes too deep and causes a stack overflow (the default limit in Python is 1000). You may get bonus points for pointing this out to the interviewer. Recursion will never be O(1) space complexity because a stack is involved, unless there istail-call optimization (TCO). Find out if your chosen language supports TCO.

Recursive functions can also be more difficult than iterative algorithms to read and debug because it can be harder to follow what is happening in a recursive algorithm.

# Applications of Recursion:

* **Tree and graph traversal**: Recursion is frequently used for traversing and searching data structures such as trees and graphs. Recursive algorithms can be used to explore all the nodes or vertices of a tree or graph in a systematic way.
* **Sorting algorithms**: Recursive algorithms are also used in sorting algorithms such as quicksort and merge sort. These algorithms use recursion to divide the data into smaller subarrays or sublists, sort them, and then merge them back together.
* **Divide-and-conquer algorithms**: Many algorithms that use a divide-and-conquer approach, such as the binary search algorithm, use recursion to break down the problem into smaller subproblems.
* **Fractal generation**: Fractal shapes and patterns can be generated using recursive algorithms. For example, the Mandelbrot set is generated by repeatedly applying a recursive formula to complex numbers.
* **Backtracking algorithms**: Backtracking algorithms are used to solve problems that involve making a sequence of decisions, where each decision depends on the previous ones. These algorithms can be implemented using recursion to explore all possible paths and backtrack when a solution is not found.
* **Memoization**: Memoization is a technique that involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization can be implemented using recursive functions to compute and cache the results of subproblems.