**Recursion**

martes, 9 de mayo de 2023

4:56 p. m.

Recursion

Recursion Intro - b001: [Recursion in Programming Explained..](https://www.youtube.com/watch?v=DUC1qg7ZaRg)

Real Python ref: <https://realpython.com/python-recursion/>

Thinking Recursively - RealPython: <https://realpython.com/python-thinking-recursively/>

Programiz ref: <https://www.programiz.com/python-programming/recursion>

geeks for geeks ref: <https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/>

w3resource - Recursion Exercises: <https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-recursion.php>

Introduction

Recursion is a fundamental concept in computer programming, and it involves solving problems by breaking them down into smaller, similar subproblems. In essence, recursion is a process where a function calls itself in order to achieve a specific task or solve a problem. This technique can be powerful and elegant, but it's important to understand how it works and when to use it effectively. Here are some key components of recursion:

Base Case: Every recursive function should have a base case, which is a condition that determines when the recursion should stop. Without a base case, the recursion would continue indefinitely and lead to a stack overflow.

Recursive Case: In addition to the base case, a recursive function also has a recursive case. This is where the function calls itself with modified arguments, making the problem smaller in some way. The recursive case helps progress toward the base case.

Recursion is about Divide and Conquer, where a complex problem is divided into smaller, manageable subproblems. Each subproblem is solved independently, and the solutions are combined to solve the original problem.

Recursion is commonly used in various programming tasks, such as traversing trees and graphs, solving mathematical problems, and more. It can lead to concise and elegant solutions but should be used with care, as excessive recursion can lead to performance issues and make the code harder to understand.

When working with recursion, it's important to plan your base cases, ensure that the recursive function makes progress toward the base case, and consider the stack depth to prevent stack overflow errors.

One tool to solve recursions is to draw the recursion tree. *Check this later / Ask ChatGPT*

refs:

* [Understanding Recursion With Trees | Trees and Recursion | Recursive Data Structures | Geekific](https://www.youtube.com/watch?v=yV48gGo5Mkc)
* [Recursion Tree Method](https://www.youtube.com/watch?v=0D2-sYen23E)
* [Introduction to recursion trees](https://www.youtube.com/watch?v=3pK89m8lNp0)

Most simple recursion cases

There are two essential cases to start understanding recursions: Factorial and Fibonacci's. Since both could be solve recursively doing one single operation and backtracking to go to the result.

From this two exercises, what is becoming evident is that in order to start to grasp how the code will flow and return what's desired, the first step is to think from the base case toward the final result.

Intuitively, the regular programming thinking is prone to start from the "beginning of the loop" and iterate toward the base case and then build back up, but while doing so is quite hard to follow the execution thread, so far, my experience tells me that is better to think from the base case, which will be when the recursion is going to stop, and then write the rest of the code accordingly.

The factorial case

This one is the most essential of the two, since there only one operation and only one condition for the recursion to stop.

# Base case

def factorial (n):

    if n == 0: return 1

    return n \* factorial(n-1)

The logic is that the code will go, leaving behind a pending operation: ( n \* *something* ). That path will lead to a point which n reaches it minimum value (0) and the will backtrack recurring to the from the "bottom" to the operations leaved pending, so would be something like:

n = 4

F(4) = 4 \* F(3)

F(3) = 3 \* F(2)

F(2) = 2 \* F(1)

F(1) = 1 \* F(0)

F(1) = 1 Here, the base condition is met: if n == 0: return 1

And from here the backtrack begins.

F(1) = 1 \* 1 = 1

F(2) = 2 \* 1 = 2

F(3) = 3 \* 2 = 6

F(4) = 4 \* 6 = 24 Arriving to the expected result.

This way of representing the case is called "Drawing the Recursion Tree". which strictly would look as follows:

factoriaIC4) 
L factoriaI(3) 
L 
fac ( P ) 
factorial(l) 
L 
factorial (O) 
Result: 
L Rosult : 
L Result : 

The Fibonacci's case

This one is the second essential case to understand recursion and seems similar to the last one but has a little trick in it.

First, since the Fibonacci's sequence is defined as the sequence of numbers resulting from summing up the last two numbers in the sequence, this suppose that the sequence has two exceptions for the first 2 numbers: 0 & 1. From there the 3rd number in the sequence would be the sum of those two, meaning 1 again, the 4th would be the sum of the next last two (1 & 1), meaning 2. And the 5th one would be the sum of the next last two (1 & 2), meaning 3, and so on.

1st 2nd 3rd 4th 5th …

0 1 1 2 3 …

Since for the first two values there is not two to be summed up, from the 3rd value the recursion really starts, consequentially, there are two base cases, when n = 0 and n = 1, to effectively return the two values needed to complete the operation and start to backtrack.

# Base algorithm

def fibonacci(n):

    if n == 0: return 0

    if n == 1: return 1

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

The logic of the recursion is the same, but the thing here is that from the very beginning two recursions start separately, one recurring to the function in n-2 and the other one in n-1 and here the flow gets a little trickier than the previous example.

Here is The Recursion Tree for n = 4 :

Fibonacci (4) 
Fibonacci(3) = 2 
FibonaGGi(2) — 
I Fibonacci(l) = 
Fi bonacci(O) — O 
L Fibonacci(l) 
Fibonacci(2) = 
Fibonacci(l) 
Fibonacci(0) = O 

Since this logic is not intuitively traceable, here is my version of how the function will run the recursion.

ÇlbaaCCl's Cage 
n-1•1 

In my version of the recursion tree, for n = 5, is easier to understand that the function will go first for the left side of the tree and then up, until it reaches the point where the right side is needed to be solved in order to return the expected final result. so in conclusion it will first solve the left side of the tree and then the right one, and the same is true for each subtree in the case.

More on recursion

Real Python ref: <https://realpython.com/python-recursion/>

Most programming problems are solvable without recursion. So, strictly speaking, recursion usually isn’t necessary.

recursion isn’t for every situation. Here are some other factors to consider:

* For some problems, a recursive solution, though possible, will be awkward rather than elegant.
* Recursive implementations often consume more memory than non-recursive ones.
* In some cases, using recursion may result in slower execution time.

Typically, the readability of the code will be the biggest determining factor.

As seen in the next example, there's time when a Non-recursive function could be harder to follow than a recursive one.

say we want to traverse through a Nested List.

names = ["Adam", ["Bob", ["Chet", "Cat"], "Barb", "Bert"], "Alex", ["Bea", "Bill"], "Ann"]

The recursive function to do so, would be the next one:

def item\_counter(item\_list):

    print(f"List: {item\_list}")

    count = 0

    for item in item\_list:

        if isinstance(item, list):

            print('Encountered Sublist')

            count += item\_counter(item)

        else:

            print(f"Counted leaf item '{item}'")

            count += 1

    print(f"-> Returning count: {count}")

    return count

And, an Iterative (Non-recursive) one, would be looking like this:

def item\_counter(item\_list):

    count = 0

    stack = list()

    element = item\_list

    i = 0

    while True:

        if i == len(element):

            if element == item\_list:

                return count

            else:

                element, i = stack.pop()

                i += 1

                continue

        if isinstance(element[i], list):

            stack.append([element, i])

            element = element[i]

            i = 0

        else:

            count += 1

            i += 1

While is not NASA science, it is certainly harder to follow the flow of the Iterative one.

|  |  |  |
| --- | --- | --- |
| SR No. | Recursion | Iteration |
| 1) | Terminates when the base case becomes true. | Terminates when the condition becomes false. |
| 2) | Used with functions. | Used with loops. |
| 3) | Every recursive call needs extra space in the stack memory. | Every iteration does not require any extra space. |
| 4) | Smaller code size. | Larger code size. |

Case - Tower of Hanoi

ref: [Towers of Hanoi: A Complete Recursive Visualization](https://www.youtube.com/watch?v=rf6uf3jNjbo)

The tower of Hanoi is a simple yet quite logically complex game that illustrates perfectly the logic of recursion.

Logotipo

Descripción generada automáticamente

Interactive version of the game: <https://www.mathsisfun.com/games/towerofhanoi.html>

The task is to move n disks from one rod to other with just two rules: Only one disk could be moved at a time and no bigger disk can be above a smaller one.

In general, The spirit of the recursion is the following:

Recursive Problem Solving 
Let f (n) be a recursive function 
1) Show f(l) works (base case) 
2) Assume f (n — l) works 
3) Show f (n) works using .f(n — 1) 

And, by following this logic is possible to solve a high logic complexity case, with a few steps:

pm(start, end) if I 
Other 6 — (start + end) 
hanoi(n, start, end) 
hanoi(n — l, start, other) 
pm(start, end) 
hanoi(n — l, other, end) 
pm(start, end) 
:= print(start, 4, end) 

Note: in the "other = 6 - (start + end)" sentence, 6 is the sum of 1 + 2 + 3, which are the names of the rods, so the way to now the destination rod or auxiliary rod is to subtracts whatever be 'start' and 'end' giving the resulting rod.

Strengths and Weaknesses

**Strengths**

**Simplicity and Readability:** Recursive solutions often mirror the natural structure of the problem, making the code more intuitive and easier to understand.

**Modularity:** Recursive functions break down a problem into smaller, more manageable sub-problems. This modularity can lead to more organized and maintainable code.

**Elegance:** In some cases, recursive solutions can be more elegant and concise than their iterative counterparts, especially for problems that exhibit a recursive structure.

**Applicability to Certain Problems:** Recursive approaches are well-suited for problems that can be naturally expressed in a divide-and-conquer manner.

**Solving Complex Problems:** Recursive algorithms are particularly useful for solving problems with a self-similar structure or problems that can be broken down into smaller instances of the same problem.

**Weaknesses**

**Performance Overhead:** Recursive solutions can have a higher memory overhead due to the function call stack. Some languages and environments might not optimize tail recursion, leading to potential stack overflow issues.

**Potential for Infinite Recursion:** If not implemented carefully, recursion can lead to infinite loops, causing the program to crash or hang.

**Difficulty in Debugging:** Debugging recursive functions can be more challenging than debugging iterative solutions. Understanding the sequence of recursive calls and their impact on the program state can be complex.

**Efficiency Concerns:** Recursive solutions may not be the most efficient for certain problems, and iterative solutions might outperform them in terms of speed and memory usage.

Recursion Use Cases

Tree Traversal: Recursive algorithms are often used to traverse tree structures, such as in binary tree and graph traversal.

Sorting Algorithms: Some sorting algorithms, like QuickSort and MergeSort, use recursion to divide the problem into smaller sub-problems.

Mathematical Calculations: Recursion is often used in mathematical calculations, such as computing factorials, Fibonacci numbers, and combinations.

Backtracking Algorithms: Problems that involve exploring all possible solutions, like the N-Queens problem or the Sudoku solver, often use recursive backtracking.

Dynamic Programming: Many dynamic programming problems involve solving sub-problems, and recursion is a natural fit for such scenarios.

Graph Algorithms: Recursive approaches can be used in graph algorithms, such as depth-first search (DFS).

Divide and Conquer: Problems that can be solved by breaking them into smaller, similar sub-problems, like the merge step in MergeSort, often benefit from recursion.