recursion_lecture_notebook

January 5, 2024

1 Introduction to Recursion

Recursion is a fundamental concept in computer science and programming. It's a problem-solving approach where a function solves a problem by calling itself with modified input, gradually breaking down complex problems into simpler sub-problems. Today, we'll dive deep into the world of recursion to understand its principles and applications.

1.1 Computing factorials

Let's start with a simple example to get the idea of recursion. Consider the problem of calculating the factorial of a number. Factorial, denoted by `n!`, is the product of all positive integers from 1 to `n`. For example, `4!` (read as ``4 factorial'') is calculated as follows: $4! = 4 \times 3 \times 2 \times 1 = 24$

Let's define a function `f(n)` as a mathematical function that calculates the factorial:

$$f : n \to n!$$

Now, let's manually unfold this function using the defined signature $f : n \to n!$ to understand how it computes the factorial of 4:

Let's unfold `f(4)` to see all the recursive calls and the smaller factorial problems:

```
f(4) = 4 * f(3)
           f(3) = 3 * f(2)
                      f(2) = 2 * f(1)
                                 f(1) = 1 * f(0)
                                            f(0) = 1
```

So, `f(4)` recursively calls `f(3)`, which calls `f(2)`, which calls `f(1)`, which calls `f(0)`. Each of these calls contributes to the final result:

1. `f(0)` is evaluated and passed to `f(1)`: `shell f(4) = 4 * f(3) f(3) = 3 * f(2) f(2) = 2 * f(1) f(1) = 1 * 1`
2. `f(1)` can now be evaluated and passed to `f(2)`: `shell f(4) = 4 * f(3) f(3) = 3 * f(2) f(2) = 2 * 1`
3. `f(2)` can now be evaluated and passed to `f(3)`: `shell f(4) = 4 * f(3) f(3) = 3 * 2`
4. `f(3)` can now be evaluated and passed to `f(4)`: `shell f(4) = 4 * 6`

5. `f(4)` can now be evaluated: `shell  f(4) = 24`

Therefore, `f(4) = 4! = 24`, matching the result we obtained earlier.

Notice how `f` first unfolds/chains a number of recursive calls (`f(3)`, `f(2)`, `f(1)`) until a base case `f(0)`, and later evaluates these from the last call:

```
f(4)
= 4 * f(3)
= 4 * (3 * f(2))
= 4 * (3 * (2 * f(1)))
= 4 * (3 * (2 * (1 * f(0))))
= 4 * (3 * (2 * (1 * 1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2))
= 4 * 6
= 24
```

Now that we've explored how the factorial function `f(n)` unfolds to calculate the factorial of a number, let's define it formally. The definition of `f` consists of two logical components:

1. **Base Case for `f(0)`:** We define the factorial of 0 as 1. This is the simplest case, where we don't need to perform any further recursion. It serves as the termination condition.

$$f : 0 \to 1$$

2. **Recursive Case:** For any positive integer `x`, we calculate `f(x)` by multiplying `x` with the result of `f(x - 1)`. This recursive step reduces the problem to a smaller subproblem, gradually reaching the base case.

$$f : x \to x \times f(x-1)$$

In Python, we can implement the formal definition of `f` as follows:

[1]:
```python
def factorial(n):
    if n == 0:
        return 1  # Base case: f(0) is defined as 1
    else:
        return n * factorial(n - 1)  # Recursive case: f(n) = n * f(n-1)

print(f"factorial({4}) = {factorial(4)}")
```

```
factorial(4) = 24
```

We can confirm that `factorial` unfolds as the formal definition of `f` by adding some debugging logic

[2]:
```python
def factorial_debug(n):
    if n == 0:
        print(f"f({n}) = 1")
        return 1  # Base case: f(0) is defined as 1
    else:
        print(f"f({n}) = {n} * f({n - 1})")
        return n * factorial_debug(n - 1)  # Recursive case: f(n) = n * f(n-1)
```

[3]:
```
print(f"f({4}) = {factorial_debug(4)}")
```

```
f(4) = 4 * f(3)
f(3) = 3 * f(2)
f(2) = 2 * f(1)
f(1) = 1 * f(0)
f(0) = 1
f(4) = 24
```

Although recursion is an elegant way to solve factorial problems, we can also implement it iteratively. The `factorial_iterative` function calculates the factorial by iteratively multiplying the integers from 1 to `n`

[4]:
```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

[5]:
```
print(f"f({4}) = {factorial_iterative(4)}")
```

```
f(4) = 24
```

1.2 Basic Concepts of Recursion

1.2.1 Key Concepts of Recursion

Recursion involves several key concepts:

1. **Base Case:** Every recursive problem must have a base case. This is the simplest instance of the problem that can be solved directly without further recursion. It serves as the stopping condition.
2. **Recursive Case:** In the recursive case, a function calls itself with modified input to solve a slightly smaller sub-problem. The recursive case reduces the problem toward the base case.
3. **Scopes:** Every recursive function call gives rise to a new scope, preserving the local variables' context. This mechanism enables the program to maintain and revisit the function's state as needed when the function concludes its execution.

1.2.2 Advantages of Recursion

Why should we use recursion? Recursion offers several advantages:

1. **Elegant Solutions:** Recursion often leads to elegant and concise solutions for complex problems. It simplifies code by breaking it into manageable pieces.
2. **Divide and Conquer:** It is a natural fit for problems that can be divided into smaller, similar sub-problems. This ``divide and conquer'' approach can be more efficient.
3. **Abstraction:** Recursion allows us to abstract complex problems into simple function calls, making code more understandable and maintainable.

1.2.3 Disadvantages and Pitfalls

While recursion is a powerful tool, it's not always the best choice:

1. **Performance:** Recursive functions can be less efficient due to the overhead of function calls and stack management. For some problems, iterative solutions may be faster.
2. **Stack Overflow:** Recursion can lead to a stack overflow error if not carefully managed. It's essential to ensure that the base case is reachable and that the problem size reduces with each recursive call.

1.3 The Three Laws of Recursion

To become proficient in recursion, it's crucial to grasp the three fundamental laws that govern how recursive functions work. These laws serve as the building blocks for solving problems using recursion.

1.3.1 Law 1: A Base Case

The first law of recursion is having a **base case**. This is the foundation upon which the entire recursion is built. The base case is the simplest, most elementary instance of the problem that can be solved directly without further recursion. It serves as the termination condition, preventing infinite recursion. In the case of the factorial example, this is $f : 0 \to 1$. Without base cases, recursion would become an endless loop without progress. Let's take the classic factorial problem as an example. The base case for calculating the factorial of 0 is defined as 1. This is the simplest instance, and it allows us to stop the recursion:

[6]:
```
def factorial_no_base_case(n):
    print(f"f({n}) = {n} * f({n - 1})")
    return n * factorial_no_base_case(n - 1)
```

[7]:
```
# As invoking factorial_no_base_case will lead to an infinite number of␣
 ↪recursive call, for demo purposes we temporarily set the limit to 50 calls.␣
 ↪The default is 1000.
import sys
sys.setrecursionlimit(50)

print(f"factorial_no_base_case(4) = ")
try:
   factorial_no_base_case(4)
except RecursionError as recursive_error:
    print(f"RecursionError: '{recursive_error}'")

sys.setrecursionlimit(1000)
```

```
factorial_no_base_case(4) =
f(4) = 4 * f(3)
f(3) = 3 * f(2)
f(2) = 2 * f(1)
f(1) = 1 * f(0)
f(0) = 0 * f(-1)
```

```
f(-1) = -1 * f(-2)
f(-2) = -2 * f(-3)
f(-3) = -3 * f(-4)
f(-4) = -4 * f(-5)
f(-5) = -5 * f(-6)
f(-6) = -6 * f(-7)
f(-7) = -7 * f(-8)
f(-8) = -8 * f(-9)
f(-9) = -9 * f(-10)
f(-10) = -10 * f(-11)
f(-11) = -11 * f(-12)
f(-12) = -12 * f(-13)
RecursionError: 'maximum recursion depth exceeded while calling a Python object'
```

1.3.2 Law 2: Making the Problem Smaller

The second law of recursion is **making the problem smaller**. In this law, a recursive function breaks the problem into smaller, more manageable sub-problems. This reduction occurs with each recursive call, bringing the problem closer to the base case. Think of this as dismantling a complex structure piece by piece. By addressing smaller parts of the problem, you make it more approachable and easier to solve. This gradual reduction simplifies the problem-solving process. In the factorial problem, making the problem smaller means calculating `f(n)` by multiplying `n` with `f(n-1)`. The problem size decreases with each recursive call, moving closer to the base case.

1.3.3 Law 3: Combine Solutions

The third law of recursion is to **combine solutions**. In this step, we utilize the results of smaller sub-problems to build the solution for the original problem. As we return from recursive calls, we combine the solutions to construct the final answer. This step unifies the results from different parts of the problem to create the whole solution. In the factorial problem, we combine the results of smaller factorials to calculate the factorial of a larger number. Each recursive call contributes to the final product, and this combination leads to the solution.

1.4 Translating Iterative Functions into Recursive Functions: Examples

For every recursive function there exists a recursive variant, let's look at some examples

1.4.1 Example 1: Fibonacci Sequence

Calculate the nth term of the Fibonacci sequence.

[8]:
```python
def iterative_fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

[9]:
```python
def recursive_fibonacci(n):
    if n <= 1:
```

```
        return n
    return recursive_fibonacci(n - 1) + recursive_fibonacci(n - 2)
```

[10]:
```
print(f"iterative_fibonacci(7) = {iterative_fibonacci(7)}")
print(f"recursive_fibonacci(7) = {recursive_fibonacci(7)}")
```

```
iterative_fibonacci(7) = 13
recursive_fibonacci(7) = 13
```

1.4.2 Example 2: Sum of Numbers

This function calculates the sum of numbers from 1 to a given positive integer n.

[11]:
```
def iterative_sum(n):
    result = 0
    for i in range(1, n + 1):
        result += i
    return result
```

[12]:
```
def recursive_sum(n):
    if n == 1:
        return 1
    return n + recursive_sum(n - 1)
```

[13]:
```
print(f"iterative_sum(5) = {iterative_sum(5)}")
print(f"recursive_sum(5) = {recursive_sum(5)}")
```

```
iterative_sum(5) = 15
recursive_sum(5) = 15
```

1.4.3 Example 3: Powers Calculation

This function calculates the result of raising a base number to a given exponent using repeated multiplication.

[14]:
```
def iterative_power(base, exponent):
    result = 1
    for _ in range(exponent):
        result *= base
    return result
```

[15]:
```
def recursive_power(base, exponent):
    if exponent == 0:
        return 1
    return base * recursive_power(base, exponent - 1)
```

[16]:
```
print(f"iterative_power(2, 3) = {iterative_power(2, 3)}")
print(f"recursive_power(2, 3) = {recursive_power(2, 3)}")
```

```
iterative_power(2, 3) = 8
recursive_power(2, 3) = 8
```

1.4.4 Example 4: List Summation

This function calculates the sum of all elements in a list.

[17]:
```python
def iterative_list_sum(arr):
    result = 0
    for num in arr:
        result += num
    return result
```

[18]:
```python
def recursive_list_sum(arr):
    if len(arr) == 0:
        return 0
    return arr[0] + recursive_list_sum(arr[1:])
```

[19]:
```python
print(f"iterative_list_sum([1, 2, 3, 4, 5]) = {iterative_list_sum([1, 2, 3, 4,␣
↪5])}")
print(f"recursive_list_sum([1, 2, 3, 4, 5]) = {recursive_list_sum([1, 2, 3, 4,␣
↪5])}")
```

```
iterative_list_sum([1, 2, 3, 4, 5]) = 15
recursive_list_sum([1, 2, 3, 4, 5]) = 15
```

1.4.5 Example 5: Robot Antennas

We have a group of robots lined up, each identified by a number from 1 to n. The odd-numbered robots (1, 3, ...) have the standard 2 antennas. However, the even-numbered robots (2, 4, ...) are equipped with 3. Determine the total count of antennas in the robot line from 1 to n, without relying on loops or multiplication.

[20]:
```python
def iterative_count_antennas(robot_ids):
    antennas = 0
    for robot_id in robot_ids:
        if robot_id % 2 == 1:
            antennas += 2
        else:
            antennas += 3
    return antennas
```

You have 3 minutes to implement a recursive solution!

[21]:
```python
def recursive_count_antennas(robot_ids):
    ...
```

[22]:
```python
def recursive_count_antennas(robot_ids):
    if not robot_ids:
```

```
        return 0
    if robot_ids[0] % 2 == 1:
        return 2 + recursive_count_antennas(robot_ids[1:])
    else:
        return 3 + recursive_count_antennas(robot_ids[1:])
```

[23]:
```
print(f"iterative_count_antennas([1, 3, 2, 7, 8, 9]) =␣
 ↪{iterative_count_antennas([1, 3, 2, 7, 8, 9])}")
print(f"recursive_count_antennas([1, 3, 2, 7, 8, 9]) =␣
 ↪{recursive_count_antennas([1, 3, 2, 7, 8, 9])}")
```

```
iterative_count_antennas([1, 3, 2, 7, 8, 9]) = 14
recursive_count_antennas([1, 3, 2, 7, 8, 9]) = 14
```

1.4.6 Example 6: List Length

This function calculates the length of a list.

[24]:
```
def iterative_list_length(arr):
    length = 0
    for _ in arr:
        length += 1
    return length
```

[25]:
```
def recursive_list_length(arr):
    if arr == list():
        return 0
    return 1 + recursive_list_length(arr[1:])
```

[26]:
```
print(f"iterative_list_length([10, 20, 30, 40, 50]) =␣
 ↪{iterative_list_length([10, 20, 30, 40, 50])}")
print(f"recursive_list_length([10, 20, 30, 40, 50]) =␣
 ↪{recursive_list_length([10, 20, 30, 40, 50])}")
```

```
iterative_list_length([10, 20, 30, 40, 50]) = 5
recursive_list_length([10, 20, 30, 40, 50]) = 5
```

1.4.7 Example 7: List Reversal

This function reverses the order of elements in a list.

[27]:
```
def iterative_list_reverse(arr):
    reversed_list = []
    for i in range(len(arr) - 1, -1, -1):
        reversed_list.append(arr[i])
    return reversed_list

    # Pythonic Alternative:
    # return arr[::-1]
```

[28]:
```
def recursive_list_reverse(arr):
    if arr == list():
        return list()
    return [arr[-1]] + recursive_list_reverse(arr[:-1])
```

[29]:
```
print(f"iterative_list_reverse([1, 2, 3, 4, 5]) = {iterative_list_reverse([1,␣
↪2, 3, 4, 5])}")
print(f"recursive_list_reverse([1, 2, 3, 4, 5]) = {recursive_list_reverse([1,␣
↪2, 3, 4, 5])}")
```

```
iterative_list_reverse([1, 2, 3, 4, 5]) = [5, 4, 3, 2, 1]
recursive_list_reverse([1, 2, 3, 4, 5]) = [5, 4, 3, 2, 1]
```

Tail Recursive List Reversal:

A recursive function is *tail recursive* if there is no suspended computation in the evaluation stack as the function unfolds. In simpler terms, it's tail recursion if the *last thing that happens in the function is the recursive call.*

Some languages employ compilers or interpreters that can optimize (unfold) tail-recursive functions such that a deeply nested recursive call won't create a whole stack of frames. On the other hand, non-tail-recursive implementations may lack performance, potentially leading to a growing stack and increased risk of stack overflow, necessitating consideration of alternative approaches for deep recursion scenarios. However, Python does not optimize tail-recursive functions, so it makes no difference here.

In the example above, `recursive_list_reverse` is not tail recursive, because the addition (`+`) operation happens after the recursive call, so the recursive call is *not* the last thing that happens. The list reversal function can be implemented relying on tail recursion by updating the output value directly in the function argument. In this approach, an additional parameter, often named `accumulator`, is used to accumulate the reversed list.

When the function is called recursively, the accumulator is updated by adding the current element to the beginning of the reversed list. This function is now tail recursive, because the addition (`+`) happens before the recursive call, which is the last thing that happens.

This implementation works by progressively building the reversed list in the accumulator, effectively reversing the order of elements in the original list. It demonstrates an alternative recursive strategy that relies on tail recursion, providing insights into how the function's evaluation unfolds.

[30]:
```
def recursive_list_reverse_tail(arr, reversed_list=None):
    if reversed_list is None:
        reversed_list = []

    if not arr:
        return reversed_list

    return recursive_list_reverse_tail(arr[:-1], reversed_list + [arr[-1]])
```

[31]:
```
print(f"recursive_list_reverse_tail([1, 2, 3, 4, 5]) =␣
↪{recursive_list_reverse_tail([1, 2, 3, 4, 5])}")
```

```
recursive_list_reverse_tail([1, 2, 3, 4, 5]) = [5, 4, 3, 2, 1]
```

5 minutes: implement `recursive_count_antennas` without using tail recursion!

[32]:
```python
def recursive_count_antennas_tail(robot_ids, total_antennas=0):
    ...
```

[33]:
```python
def recursive_count_antennas_tail(robot_ids, total_antennas=0):
    if not robot_ids:
        return total_antennas
    current_robot_id = robot_ids[0]
    if current_robot_id % 2 == 1:
        return recursive_count_antennas_tail(robot_ids[1:], total_antennas + 2)
    else:
        return recursive_count_antennas_tail(robot_ids[1:], total_antennas + 3)
```

[34]:
```python
print(f"     recursive_count_antennas([1, 3, 2, 7, 8, 9]) =␣
 ↪{recursive_count_antennas([1, 3, 2, 7, 8, 9])}")
print(f"recursive_count_antennas_tail([1, 3, 2, 7, 8, 9]) =␣
 ↪{recursive_count_antennas_tail([1, 3, 2, 7, 8, 9])}")
```

```
    recursive_count_antennas([1, 3, 2, 7, 8, 9]) = 14
recursive_count_antennas_tail([1, 3, 2, 7, 8, 9]) = 14
```

1.4.8 Example 8: String Reversal

This function reverses a string.

[35]:
```python
def iterative_string_reverse(s):
    return s[::-1]
```

[36]:
```python
def recursive_string_reverse(s):
    if s == "":
        return s
    return s[-1] + recursive_string_reverse(s[:-1])
```

[37]:
```python
print(f'iterative_string_reverse("Hello World!") =␣
 ↪{iterative_string_reverse("Hello World!")}')
print(f'recursive_string_reverse("Hello World!") =␣
 ↪{recursive_string_reverse("Hello World!")}')
```

```
iterative_string_reverse("Hello World!") = !dlroW olleH
recursive_string_reverse("Hello World!") = !dlroW olleH
```

1.4.9 Example 8: String Palindrome Check

This function checks if a string is a palindrome (reads the same forwards and backward).

[38]:
```python
def iterative_is_palindrome(s):
    return s == s[::-1]
```

[39]:
```
def recursive_is_palindrome(s):
    if len(s) == 0:
        return True
    if s[0] != s[-1]:
        return False
    return recursive_is_palindrome(s[1:-1])
```

[40]:
```
print(f'iterative_is_palindrome("radar") = {iterative_is_palindrome("radar")}')
print(f'recursive_is_palindrome("radar") = {recursive_is_palindrome("radar")}')
```

```
iterative_is_palindrome("radar") = True
recursive_is_palindrome("radar") = True
```

[41]:
```
print(f'iterative_is_palindrome(["Hello", 2, True, True, 2, "Hello"]) =␣
 ↪{iterative_is_palindrome(["Hello", 2, True, True, 2, "Hello"])}')
print(f'recursive_is_palindrome(["Hello", 2, True, True, 2, "Hello"]) =␣
 ↪{recursive_is_palindrome(["Hello", 2, True, True, 2, "Hello"])}')
```

```
iterative_is_palindrome(["Hello", 2, True, True, 2, "Hello"]) = True
recursive_is_palindrome(["Hello", 2, True, True, 2, "Hello"]) = True
```

1.4.10 Example 9: Counting Occurrences

This function counts the occurrences of a character in a string.

[42]:
```
def iterative_count_occurrences(s, char):
    count = 0
    for c in s:
        if c == char:
            count += 1
    return count
```

3 minutes: implement the recursive equivalent of `iterative_count_occurrences`!

[43]:
```
def recursive_count_occurrences(s, char):
    ...
```

[44]:
```
def recursive_count_occurrences(s, char):
    if s == "":
        return 0

    # Alternative 1:
    # if s[0] == char:
    #     return 1 + recursive_count_occurrences(s[1:], char)
    # return recursive_count_occurrences(s[1:], char)

    # Alternative 2:
    return (s[0] == char) + recursive_count_occurrences(s[1:], char)
```

[45]:
```
print(f'iterative_count_occurrences("Hello World!", "l") =␣
 ↪{iterative_count_occurrences("Hello World!", "l")}')
print(f'recursive_count_occurrences("Hello World!", "l") =␣
 ↪{recursive_count_occurrences("Hello World!", "l")}')
```

```
iterative_count_occurrences("Hello World!", "l") = 3
recursive_count_occurrences("Hello World!", "l") = 3
```

1.4.11 Example 10: Character Removal

This function removes all occurrences of a character from a string.

[46]:
```
def iterative_remove_character(s, char):
    return s.replace(char, "")
```

[47]:
```
def recursive_remove_character(s, char):
    if s == "":
        return s
    if s[0] == char:
        return recursive_remove_character(s[1:], char)
    return s[0] + recursive_remove_character(s[1:], char)
```

[48]:
```
print(f'iterative_remove_character("Hello, World!", "l") =␣
 ↪{iterative_remove_character("Hello, World!", "l")}')
print(f'recursive_remove_character("Hello, World!", "l") =␣
 ↪{recursive_remove_character("Hello, World!", "l")}')
```

```
iterative_remove_character("Hello, World!", "l") = Heo, Word!
recursive_remove_character("Hello, World!", "l") = Heo, Word!
```

1.5 Recursions & Recursive Data Structures

In addition to recursive functions, we encounter another concept: **recursive data structures**. These are data structures that contain instances of the same type within themselves, creating a self-referential, hierarchical structure.

The Basic Node Structure To understand recursive data structures, let's examine a classic example: the **linked list**. A linked list is a data structure made up of nodes, where each node contains data and a reference (or a pointer) to the next node in the list.

Motivation for LinkedLists:

Imagine you are organizing a collection of books on a shelf, and you decide to use a traditional array. Arrays are like a row of numbered book slots -- each book has its designated place, making it easy to find them quickly.

Now, consider a LinkedList as a string of interconnected bookmarks. Each bookmark points to the next one, forming a linked chain. This brings some advantages:

1. **Flexibility in Size:** Unlike arrays that need a fixed space upfront, LinkedLists can dynamically grow or shrink. Imagine inserting a new book -- with bookmarks, you can easily slide it between existing books without rearranging everything.
2. **Ease of Insertion and Deletion:** LinkedLists shine when you need to add or remove books frequently. No need to shift all the books; just update the bookmarks, and you're done.
3. **Memory Efficiency:** LinkedLists can be more memory-friendly. In an array, you might allocate space for 100 books even if you have just 10. LinkedLists use memory only for the books you actually have.

However, there are trade-offs:

1. **Random Access Limitation:** If you want to find the book at position 50, arrays are like an index where you jump directly to it. LinkedLists, on the other hand, require flipping through the bookmarks from the beginning.
2. **Memory Overhead:** Each bookmark in a LinkedList carries extra information (the address of the next bookmark). While this makes them flexible, it comes at a small cost in terms of memory.

So, LinkedLists provide a dynamic, flexible way to organize data, especially when you prioritize easy insertion and deletion over quick random access -- a bit like managing your bookshelf with interconnected bookmarks.

[49]:
```
class Node:
    def __init__(self, data, next_node=None):
        self.data = data
        self.next_node = next_node
```

We can create a LinkedList like so:

[50]:
```
fruits = Node("Apple", Node("Orange", Node("Pear")))
```

Recursive data structures, like LinkedLists, can be traversed recursively. For example, we can define our `print` function called `print_linked_list`. This function is recursive, accepts a node as its input and prints the entire linked list to standard output. It obeys to the 3 laws accordingly: 1. **Base Case**: if a NULL node is give, then the linked list is empty and the routine terminates 2. **Making the Problem Smaller**: the function prints the node given to standard output 3. **Combine Solution**: the function continues to print the next node

[51]:
```
def print_linked_list(node):
    # Base-case: an empty list.
    if node is None:
        return

    # Smaller problem: print one item.
    print(node.data, end=", ")
```

```
    # Inductive case: print the next.
    print_linked_list(node.next_node)
```

[52]:
```
print_linked_list(fruits)
```

```
Apple, Orange, Pear,
```

we can rewrite the same logic in iterative form:

[53]:
```
def print_linked_list_iterative(node):
    tmp = node
    while tmp is not None:
        print(tmp.data, end=", ")
        tmp = tmp.next_node
```

[54]:
```
print_linked_list_iterative(fruits)
```

```
Apple, Orange, Pear,
```

Similarly, we can compute the length of a linked list with a function for which: 1. **Base Case**: if a NULL node is give, then the linked list is empty and the function returns 0 2. **Making the Problem Smaller**: the function counts the node given 3. **Combine Solution**: the function continues to count the next node

[55]:
```
def length_linked_list(node):
    # Base-case: an empty list.
    if node is None:
        return 0

    # Smaller problem and Inductive case
    return 1 + length_linked_list(node.next_node)
```

[56]:
```
length_linked_list(fruits)
```

[56]:
```
3
```

we can rewrite the same logic in iterative form:

[57]:
```
def length_linked_list_iterative(node):
    length = 0
    tmp = node
    while tmp is not None:
        length += 1
        tmp = tmp.next_node
    return length
```

[58]:
```
length_linked_list_iterative(fruits)
```

[58]:
```
3
```

More Examples: Trees and Graphs Recursive data structures are not limited to linked lists. They are prevalent in other data structures, such as trees and graphs.

Trees A binary tree is a hierarchical structure where each node has at most two children, forming a tree of subtrees. Recursion is often used to traverse and manipulate binary trees.

Graphs Graphs can be defined recursively when nodes connect to other nodes, creating subgraphs. Graph traversal and search algorithms often utilize recursion.

1.5.1 Example: Divide and Conquer with File Folder Trees

Introduction to the Problem Imagine you're managing a large folder structure on your computer. You want to find and list the contents of a specific file path, such as ``/home/documents/project/report.txt.'' Searching through all folders one by one can be time-consuming. This is where the Divide and Conquer approach comes in handy.

Applying Divide and Conquer Let's break down the problem using Divide and Conquer:

1. **Divide:** We start at the root folder and divide the problem into smaller subproblems. Each subproblem involves navigating into a subfolder or file. In this way, we traverse down the folder tree.
2. **Conquer:** When we reach a folder, we recursively apply the same Divide and Conquer process within that folder. If we encounter a file, we note its name. This step repeats until we find the desired file or folder.
3. **Combine:** The results of the subproblems are combined to produce the final list of contents. Each time we traverse deeper into a subfolder, we maintain a list of contents encountered so far.

[59]:
```python
import abc

class FileSystemItem(abc.ABC):
    def __init__(self, name: str):
        self.name = name


class Folder(FileSystemItem):
    def __init__(self, name: str):
        super().__init__(name)
        self.items = dict()


class File(FileSystemItem):
    def __init__(self, name: str, content: str):
        super().__init__(name)
        self.content = content
```

Let's use this abstraction to represent the following file structure:

```
home/
 |- desktop/
 |   |- report.txt -> "Report text"
 |- documents/
 |   |- notes.txt -> "Notes text"
```

[60]:
```
report_file = File("report.txt", "report text")
notes_file = File("notes.txt", "notes text")

desktop_folder = Folder("desktop")
desktop_folder.items[report_file.name] = report_file

documents_folder = Folder("documents")
documents_folder.items[notes_file.name] = notes_file

home_folder = Folder("home")
home_folder.items[desktop_folder.name] = desktop_folder
home_folder.items[documents_folder.name] = documents_folder

root_folder = Folder("/")
root_folder.items[home_folder.name] = home_folder
```

Let's create a function to print the content of a filepath. This takes as input a FileSystemItem and a list of file system item names thank point to the file, and print the content of the file to standard output:

[61]:
```
def print_file_content(file_system_item, path) -> None:
    # Base case.
    if len(path) == 0:
        print(file_system_item.content)
        return

    # Smaller problem and Combine.
    next_name = path[0]
    tail_path = path[1:]
    next_file_system_item = file_system_item.items[next_name]
    print_file_content(next_file_system_item, tail_path)
```

[62]:
```
print_file_content(root_folder, ["home", "documents", "notes.txt"])
```

```
notes text
```

[63]:
```
print_file_content(root_folder, ["home", "desktop", "report.txt"])
```

```
report text
```