Recursion and Memoization

Recursion

In recursive algorithms, a function calls itself with a simplified version of the original problem until a base case is reached.

- it is important to have a clear understanding of the base case (the point at which the recursion will stop)
- the logic for breaking the problem down into smaller subproblems that can be solved recursively.

Recursion – internal operation

- In a computer, the memory stores the instructions and data for a program.
- When a program performs a recursive call,
  - the memory stores the state of the program at the time of the call, including the values of any variables and the location in the code that the program is about to execute.
  - This information is stored on the **call stack,** which is a data structure used by the operating system to manage the execution of a program.

Recursion – Call stack

When a program makes a recursive call, **a new stack frame** is created on the call stack to store the state of the program at the time of the call.

- **Each stack frame corresponds to** a **single function call** and includes information
  - the values of the function's arguments
  - the location in the code where the function was called from
  - and the values of the function's local variables.

Recursion – return from the recursive call

When the function returns from the recursive call -

- the state stored in the corresponding stack frame is discarded,
- and the program continues executing where it left off before the call.
- If the function makes another recursive call, another stack frame is created, and so on.

Recursion – Stack overflow

The call stack grows and shrinks dynamically as the program makes and returns from function calls.

- If the program makes too many recursive calls, the call stack can overflow, causing a runtime error called a stack overflow.
- There are some programming language specific prevention mechanism such as “tail call optimization” -- has its own issues, thus not a universal solution

Memoization in Recursion

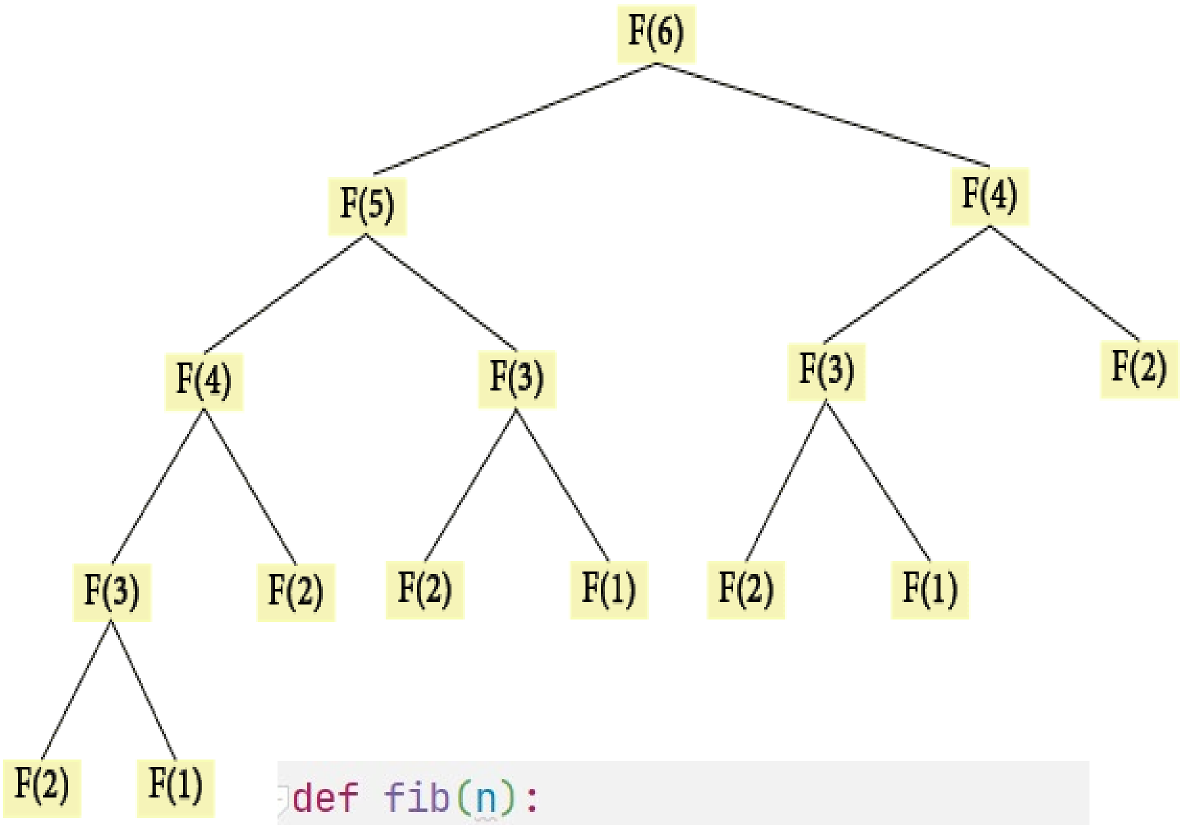
- Memoization is a technique used to optimize the performance of recursive algorithms.
  - saving the results of expensive function calls and returning the cached results when the same inputs occur again.
  - This avoids repeating the same calculation, reducing the time complexity of the algorithm.
  - useful in cases where the recursive function is called multiple times with the same inputs

Memoization in Recursion

Not a general solution?

- For instance, in the case of sorting algorithms, the benefit of memoization is limited because
  - the calculations performed by the algorithms are not repeated with the same inputs, making it difficult to make use of memoization in a straightforward way.

Memoization in Recursion (cont.)




```python
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

fib(3)
```

```python
term = [0 for i in range(100)]
# Fibonacci Series using memoized Recursion
def fib(n):
    # base case
    if n <= 1:
        return n
    # if fib(n) has already been computed,
    # it reduces the number of repeated  work
    if term[n] != 0:
        return term[n]

    else:
        # store the computed value of fib(n)
        # in an array term at index n to
        term[n] = fib(n - 1) + fib(n - 2)
        return term[n]
# Driver Code
n = 6
print("The Fibonacci series up to the", n, "th term:")
for i in range(n):
    print(fib(i), end=", ")
```

```
The Fibonacci series up to the 6 th term:
0, 1, 1, 2, 3, 5,
```

Complexity Issues – Computation Cost

- The normal version, without memoization, has an exponential time complexity of ***O(2^n)***.
  - the number of calculations needed to find the nth term in the series grows exponentially as ***n*** increases.
  - This can lead to a very slow and inefficient calculation for large values of n.

- The memoized version, on the other hand, has a time complexity of ***O(n)***.
  - This is because the results of previously computed terms in the series are stored in an array, reducing the number of repeated calculations.
  - As a result, the number of calculations needed to find the nth term grows linearly with n, making the calculation much faster and more efficient.