MCT-243 : COMPUTER PROGRAMMING-II

using Python 3.9

Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

https://www.youtube.com/playlist?list=PLWF9TXck7O_wMDB-VriREZ6EvwkWLNB7q

©Department of Mechatronics and Control Engineering, University of Engineering and Technology, Lahore, Pakistan

Lab 28: RECURSIVE FUNCTIONS & MEMOIZATION

Recursive Functions:

A function that calls itself is known as **Recursive** function. It means that the function will call itself and that will again call itself and so on. A simple recursive function to display all integers starting from input value to all next integers is given here:

```
def numberSequence(n):
    print(n)
    numberSequence(n+1)

### Main Program starts from here ###
x=eval(input("Enter starting number: "))
numberSequence(x)
```

If you run the above code, you will see that next integers will keep displaying on the screen until the final limit of self-recursion is reached.
In recursive functions we must have some stopping criteria that will not further call itself. This is also known as the **base case**. In above example if we want to display integers from entered number to 100 only, it can be done as:

```
def numberSequence(n):
    print(n)
    if(n<100):  #Base Case
        numberSequence(n+1)

### Main Program starts from here ###
x=eval(input("Enter starting number: "))
numberSequence(x)
```

Now let's see one practical example of writing a recursive function and that is to find the factorial of input number. Firstly, let's see if the factorial formula can be solved recursively or not. The formula of factorial of x is:

$$x! = x(x-1)!$$

The formula indicates that it can be solved recursively since the formula of factorial involves another factorial. That another factorial i.e. $(x-1)!$ will be evaluated again using the factorial formula as:

$$x! = x(x-1)(x-2)!$$

This will continue until there is 0! and for that we don't need factorial again as 0!=1. Hence:

$$x! = x(x-1)(x-2)(x-3)(x-4)\ldots(3)(2)(1)(0!)$$

$= x(x-1)(x-2)(x-3)(x-4)\ldots(3)(2)(1)(1)$

0!=1 is the **base case** for this scenario as it doesn't further need factorial. A user-defined function based on this logic is here:

```
def fact(x):
    'It returns the factorial of input'
    if(x==0):       # Base Case
        return 1
    ans=x*fact(x-1) # Recursive Case
    return ans
### Main Program Starts Here ###
print(fact(6))
```

Tasks:

[1] Below is a series known as **Fibonacci Series:**

$$F(n) = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Note that after first two numbers, next number is sum of previous two. So, Fibonacci series can be described as:

$F(0) = 0$	Base Case
$F(1) = 1$	Base Case
$F(n) = F(n-1) + F(n-2)$	Recursive Case

Create a user define function with one input argument as **n** and it should return the value of Fibonacci Number of that **n**. In main program take value of **n** from user and use the function created to display result on the screen.

[2] Now create a Non-Recursive user-defined function for the above task and you will realize that recursive logic is much simpler as compared to the non-recursive approach.

If we compare the recursive and non-recursive approaches, the recursive approach is simpler to implement in many cases since the implementation logic is quite close to the actual definition of the function. But we should also compare the two approaches on the basis of their performance (time of execution). In next section, we will explore the performance and will study about the **Memoization** concept. Memoization is not juts meant for the recursive function but is also applicable for the non-recursive functions.

Concept of Memoization:

In computing, **Memoization** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Let's explore this idea by coding a Fibonacci Series given as:

$$F(0) = 1$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

First ten series elements are shown here:

Index (n)	0	1	2	3	4	5	6	7	8	9
Value (F(n))	1	1	2	3	5	8	13	21	34	55

We can code the series using **Recursive Function**:

```
def fibo_recursive(n):
    if n==0:
        return 1
    if n==1:
        return 1
    if(n>1):
        return fibo_recursive(n-1)+fibo_recursive(n-2)
```

You can test this for different inputs. But let's test it for numbers 0-99 as:

```
for i in range(100):
    print(f'{i} : {fibo_recursive(i)}')
```

If you run the above code, you will see the results on the screen but they will start taking noticeable time after few values, depending upon the computer. Why is this so?

The reason is the recursive behavior of the code. Fibonacci number for n=30, for example, is calculated as:

F(30) = F(29)+F(28)
= F(28)+F(27)+F(27)+F(26)
= F(27)+F(26)+F(26)+F(25)+F(26)+F(25)+F(25)+F(24) and so on.

You can see that function is being called thousands time (Order: O(n^2)) just to calculate F(30). However if F(29) and F(28), that were calculated earlier in the `for` loop, were saved in memory, they could have been used right away without calling the function so many times.

The above concept is illustrated in below code where we have created an empty **dictionary** and whenever there is a function call for any input, the result is returned and also stored inside the dictionary so that if the function is called for the same input later, the result will be picked from the dictionary rather than calling the function recursively.

```
fibo_dict={} #An empty dictionary (key=n , value=F(n))
def fibo_cache(n):
    if n in fibo_dict: #If output against n is availble in dictionary
        return fibo_dict[n]
    else:
        if n==0:
            ans=1
        if n==1:
            ans=1
        if(n>1):
            ans=fibo_cache(n-1)+fibo_cache(n-2)
        fibo_dict[n]=ans #Push the result in dictionary
        return ans
```

Now run the same code on this function as:

```
for i in range(100):
    print(f'{i} : {fibo_cache(i)}')
```

You should be able to see the big difference. Even if you run the loop 1000 times instead of 100, you will get the quick result.

Memoization with Function Decorator:

The last example has demonstrated the concept of memoization to save memory and the execution time. Instead of creating a dictionary and storing the function calls manually, we can do it using a **Function Decorator** known as **`lru_cache()`** (LRU= Least Recently Used) available in **`functools`** module. Function Decorators are used to add extra features to any function. The **`lru_cache`** function decorator will do the job of memoization all by itself. Default value of number of function calls this is going to store in 128 but we can change it to any value by using **`maxsize`** argument. The code with this approach is shown here:

```
from functools import lru_cache
@lru_cache(maxsize=1000)
def fibo_recursive(n):
    if n==0:
        return 1
    if n==1:
        return 1
    if(n>1):
        return fibo_recursive(n-1)+fibo_recursive(n-2)
```

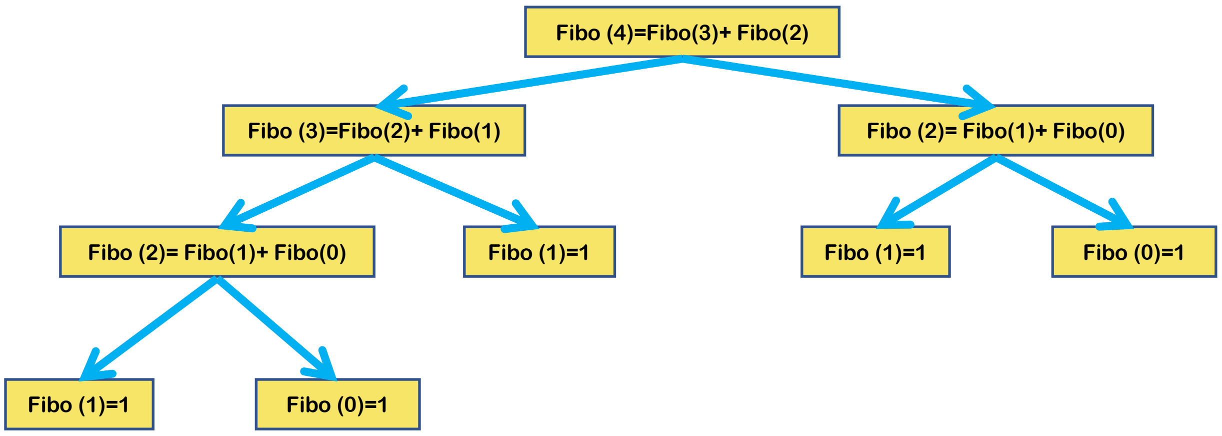
And now if we run this code, it should give results in much less time:

```
for i in range(1000):
    print(f'{i} : {fibo_recursive(i)}')
```

How much is improvement with Memoization:

In above example we are able to see a remarkable difference between the execution time of Fibonacci Function with Memoization and without Memoization. But let's try to quantify exactly how much is the difference.

If you see the logic of the Fibonacci Function you can easily see that to calculate one value, the function is called on two more values and then again two calls for each of those and so on. It will be better to see that pictorially as shown below for n=4:



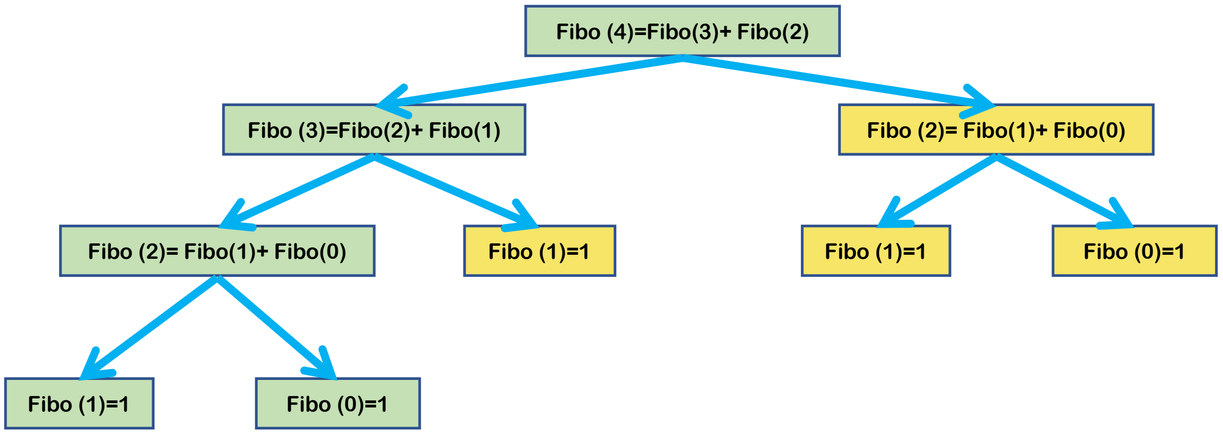
Fibo (4)=Fibo(3)+ Fibo(2)
Fibo (3)=Fibo(2)+ Fibo(1)
Fibo (2)= Fibo(1)+ Fibo(0)
Fibo (2)= Fibo(1)+ Fibo(0)
Fibo (1)=1
Fibo (1)=1
Fibo (0)=1
Fibo (1)=1
Fibo (0)=1

Can you figure out for a general value **n**, how many will be calls to the function? If for any value the function is called two times and then again two times for each of the those and so on, then the total calls for value **n** will be $\mathbf{2^n}$. You can see that the right branches in above schematic are shorter as compared to the left branches. So total calls will be less than $\mathbf{2^n}$. However, we have a parameter known as **Complexity** or the **Cost** of some algorithm. It is also known as **Order** of the algorithm and expressed as **O(n)**. This notation O(n) is also called as Big-O notation.

The order of some algorithm defines an expression which indicates that for a general value of **n**, the calculations needed by the algorithm are proportional to that expression. So, in above case the, the number of calculations is not exactly $\mathbf{2^n}$ but of course are proportional to $\mathbf{2^n}$. And hence the order of above algorithm is:

$\mathbf{O(n)= 2^n}$

Now let's explore the order of the same function when Memoization is applied. When Memoization is applied, there is no repeated calls to the same function. So in above pictorial representation, if you remove all repeated call nodes, you will left with only the right most nodes and one last node of 0. Those are indicated here with light green color:



Fibo (4)=Fibo(3)+ Fibo(2)
Fibo (3)=Fibo(2)+ Fibo(1)
Fibo (2)= Fibo(1)+ Fibo(0)
Fibo (2)= Fibo(1)+ Fibo(0)
Fibo (1)=1
Fibo (1)=1
Fibo (0)=1
Fibo (1)=1
Fibo (0)=1

For n=4, the function is called 5 times and for general value **n** the function will be called **n+1** times. And hence the order of the function with Memoization becomes **n+1** which is same as **n**. Because, as mentioned earlier, the order of a function indicates that the number of calculations are proportional to specified expression. Something proportional to **n+1** is equivalent as proportional to **n**. So, here is the comparison of the complexity of the function with Memoization and without Memoization:

O(n)= 2^n (without Memoization)

O(n)= n (with Memoization)

Evaluate the above expression for any value of **n** and you will see the difference. The order of the function of Fibonacci Series with non-recursive approach is also **n**. So with recursive function we got the advantage of simple code but that was not efficient and when applied Memoization, that became efficient too.

Monitor the execution time of a code:

Memoization improved the code timing to a great extent. We could see that clearly, however there are tools by which we can precisely monitor the execution time of a code. The function is **`timeit()`** that is available in **`timeit`** module. We pass the code as first input argument and we can also specify the number of times we want to run the code. The function will give the number of seconds took to execute the code that number of times.
A simple example code is given here:

```
import timeit
print(timeit.timeit("print('A')",number=10))
```

You will see **`A`** gets printed 10 times and then will be the number of seconds for the execution. However, if we try to run this function on our created function as shown here:

```
import timeit
def fibo_recursive(n):
    if n==0:
        return 1
    if n==1:
        return 1
    if(n>1):
        return fibo_recursive(n-1)+fibo_recursive(n-2)

print(timeit.timeit('fibo_recursive(30)',number=10))
```

Here we will get a **`NameError`**. Basically, **`timeit`** executes the code in its own namespace rather than the code's namespace. The solution is to simply assign the code's namespace to **`timeit`** namespace as:

```
print(timeit.timeit('fibo_recursive(30)',globals=globals(),number=10)
)
```

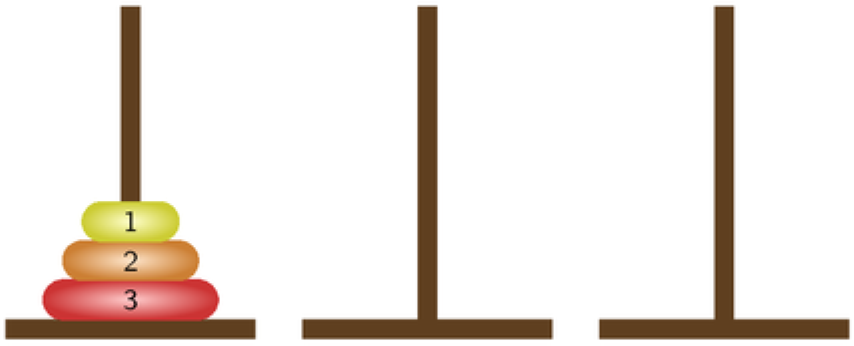
Note the time and then add **`lru_cache()`** decorator, run the code again and see the difference in time.

One last point for the Memoization is that it is not just meant for the Recursive functions. It can be used for non-recursive functions as well. Considering the Fibonacci Series again, with recursive function the Memoization was helpful even when we were evaluating that for one number e.g., n=10, because even for one number, function is called repeatedly and Memoization will improve that. But if we are evaluating the non-recursive function for n=10 there is no advantage of Memoization because no repeated function is called. However, if we are applying the non-recursive function on many numbers (repeated numbers), then will be the advantage of Memoization.

Tasks:

[1] Greatest Common Divisor (GCD): The Greatest Common Divisor (GCD) of two numbers can be found recursively as:
- Input arguments are two number `a` and `b`
- If second number is `0`, return `a`.
- Otherwise return GCD of `a` and `a%b`
Implement the GCD function with this logic.

[2] Tower of Hanoi: The **Tower of Hanoi,** is a mathematical problem which consists of three rods and multiple disks. Initially, all the disks are placed on one rod, one over the other in ascending order of size similar to a cone-shaped tower. An example with three disks is shown here:



The objective of this problem is to move the stack of disks from the initial rod to another rod, following these rules:

a) Only one disk can be moved at a time.
b) A disk cannot be placed on top of the smaller disk.

The goal is to move all the disks from the leftmost rod to the rightmost rod. To move **N** disks from one rod to another, $\mathbf{2^{N-1}}$ steps are required. So, to move 3 disks from starting the rod to the ending rod, a total of 7 steps are required.

This problem can be solved with the recursive approach for any number of disks. Check the algorithm explained in the link below and create a recursive function to solve the problem.
https://www.cs.cmu.edu/~cburch/survey/recurse/hanoiimpl.html

Recursion Error in Recursive Functions:

Let's consider the Fibonacci Series function with Memoization again and print the 499th number of the series:

```
print(fibo_recursive(499))
```

You will get this output:

```
1394232245616978801397243828704072839500702565876973072641089629483255
716228632906915576588762225212941 25
```

Now if we try to access the next number is series i.e., the 500th number, it will generate `RecursionError`.

```
print(fibo_recursive(500))
```

You will see this error:

```
RecursionError: maximum recursion depth exceeded
```

There is a maximum number of inner recursions that interpreter can handle and that is known as **Maximum Recursion Depth**.

Firstly, it is important to know why there is a limit on maximum number of inner calls to a function. Whenever a function is called, whether recursive or non-recursive, the interpreter stores some related data in Memory known as **Stack Memory**. This includes the values assigned to different local variables of the function and most importantly the return address of the function. We know, after execution of the function, the interpreter must return back to the point where the function was called. That is possible by storing the return address of that point when a function is called. If a function, calls another function, that return address will also be stored and you can imagine that in case of recursive function, there will be series of inner calls to the same function and all return addresses will be stored on the Stack Memory. This can consume a lot of memory depending upon the number of inner calls.

Therefore, Python puts a limit on the number of inner calls in case of recursive function and that is known as **Maximum Recursion Depth**.

We can see the value of this limit from the `sys` module as:

```
import sys
print(sys.getrecursionlimit())
```

You will see the output as 1000.

So, 1000 is the value of Maximum Recursion Depth. Why we got this error when we executed Fibonacci Function on 500? Because the Fibonacci recursive function involves double recursion as `return fibo_recursive(n-1)+fibo_recursive(n-2)`. And hence in case of 500, there are 1000 inner calls to the function in recursive case, one last base case plus one original call as `fibo_recursive(500)` making a total of 1002 calls that exceeds the limit and we got the error.

There is a function in `sys` module `setrecursionlimit` that we can use to set the value and if we set the limit as 1002, then for `fibo_recursive(500)`, we will get the answer instead of error.

Conclusion:

- Recursive functions are simple to implement as compared to non-recursive functions (of course if applicable).
- Recursive function can take a lot of time as compared to non-recursive functions but that can be handled by using the concept of Memoization and then the time consumption can be same or even better as compared to non-recursive functions.
- Recursive functions can take a lot of memory (stack memory) if there are a large number of inner calls. And hence these are not preferred when they will be used for large number of inner calls. For example, to find a large Fibonacci Number, like 300th or bigger, it should not be preferred but usually we do not need such big Fibonacci Number. Also, in case of the GCD and Tower of Hanoi, these functions will not have large inner calls in general and hence recursive function is preferred for such cases.