IS 4th Sem

Recursion—WEEK 9

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.

- A recursive function solves a particular problem by calling a copy of itself and solving smaller sub problems of the original problems.
- Many more recursive calls can be generated as and when required.
- It is essential to know that we should provide a certain case in order to terminate this recursion process.
- we can say that every time the function calls itself with a simpler version of the original problem.

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it.

For example;

Recursive solutions are best suited to some problems like:

1. Tree Traversals: InOrder, PreOrder, PostOrder
2. Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
3. Tower of Hanoi
4. Backtracking Algorithms
5. Divide and Conquer Algorithms
6. Dynamic Programming Problems
7. Merge Sort, Quick Sort
8. Binary Search
9. Fibonacci Series, Factorial, etc.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Algorithm: Steps

- The algorithmic steps for implementing recursion in a function are as follows:
- **Step1** - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

- **Step2** - Define a recursive case: Define the problem in terms of smaller sub problems. Break the problem down into smaller versions of itself, and call the function recursively to solve each sub problem
- **Step 3**: Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop
- **step4** - Combine the solutions: Combine the solutions of the sub problems to solve the original problem

Recursive Functions:

1.Factorial:

- Factorial of a non-negative integer is the multiplication of all positive integers smaller than or equal to n.
- For example factorial of 6 is 6*5*4*3*2*1 which is 720.
- A factorial is represented by a number and a " ! " mark at the end.
- It is widely used in permutations and combinations to calculate the total possible outcomes.
- Let's create a factorial program using recursive functions. Until the value is not equal to zero, the recursive function will call itself. Factorial can be calculated using the following recursive formula.

n! = n * (n – 1)!
n! = 1 if n = 0 or n = 1

factorial(4) #1st call with 4
4* fact(3) #2nd call with 3
4 *3* fact(2) #3rd call with 2
4 *3*2*fact(1) #4th call with 1
4*3*2*1 #return from 4th call as number =1
4*3*2 #return from 3rd call as number =2
4*6 #return from 2nd call
24 #return from 1st call

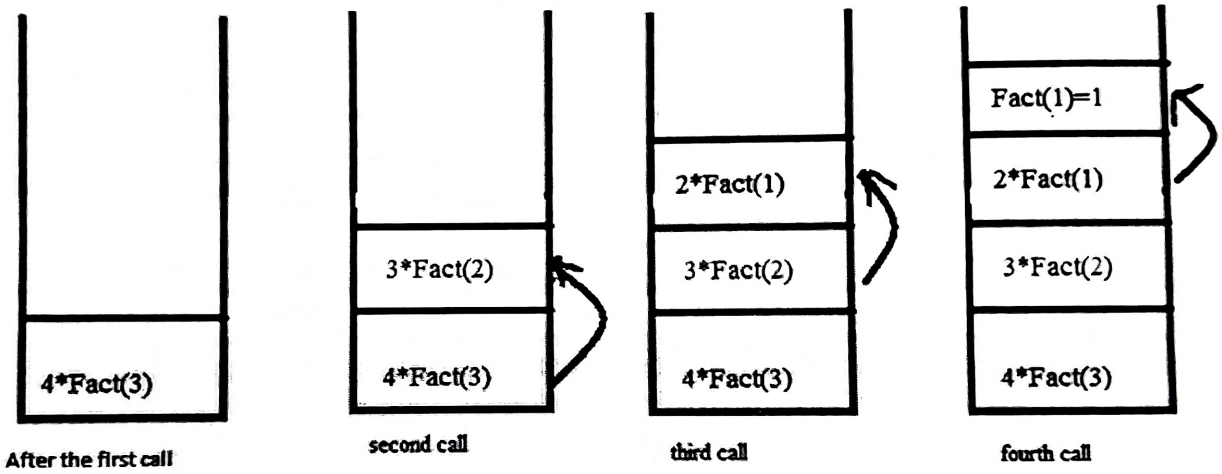
Python code:

```
def factorial(n):
    return 1 if (n == 1 or n == 0) else n * factorial(n - 1)        # single line to find factorial

num = 5                    # Driver Code

print ("Factorial of", num, "is",

    factorial(num))
```

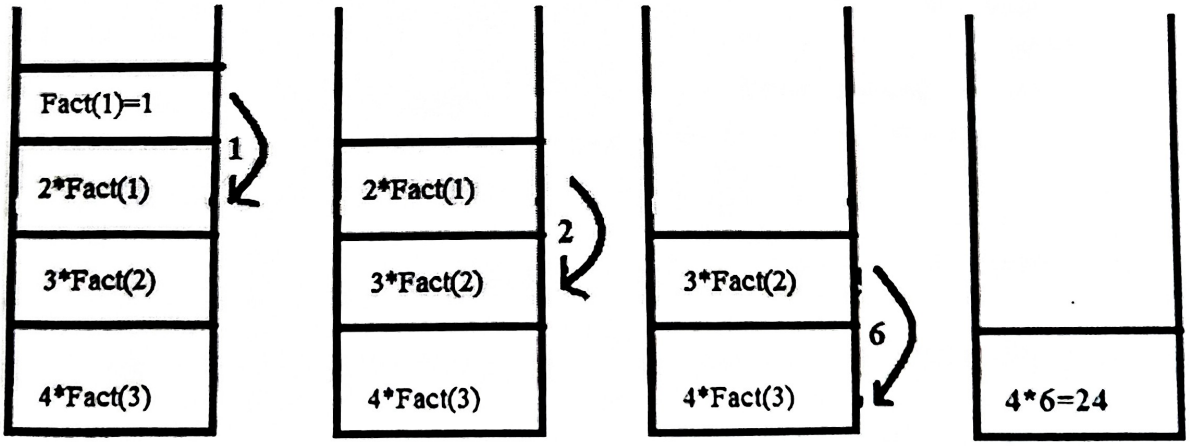
2.Call Stack Recursion

- A call stack is a data structure used by computer programs to keep track of function calls. When a function is called, a new frame is added to the call stack.
- This frame contains information about the function call, such as the function arguments and the current position in the code.
- When the function completes its execution, the frame is removed from the call stack, and the program returns to the previous frame. T
- he call stack operates in a Last-In-First-Out (LIFO) manner, which means that the last function called is the first to be completed.
- This is important to keep in mind when dealing with recursive functions, as each recursive call adds a new frame to the call stack, which can lead to stack overflow errors if not managed properly.

When function call happens previous variables gets stored in stack

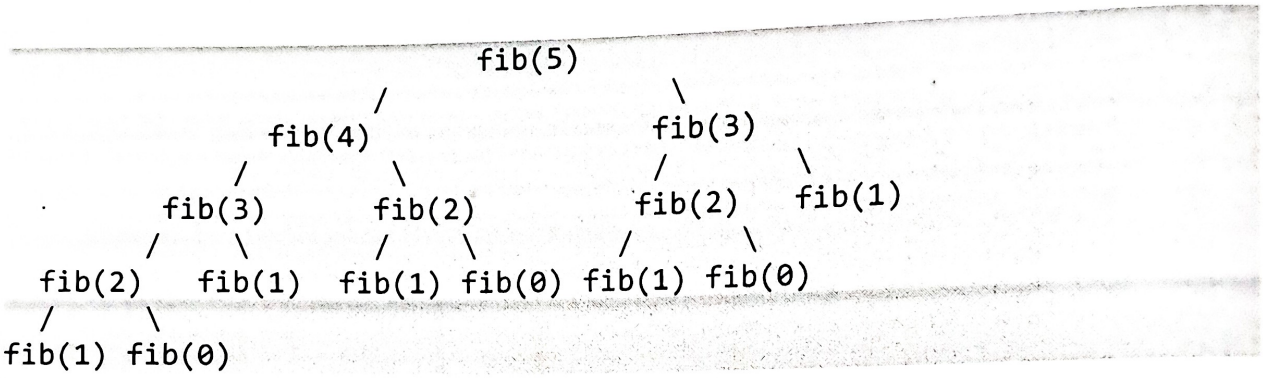


Returning values from base case to caller function



2. Fibonacci sequence: *Fibonacci sequence is described as*: the sequence of numbers where the first two numbers are 0 and 1, with each subsequent number being defined as the sum of the previous two numbers in the sequence.

The Fibonacci sequence looks like this:0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 and so on.



Python code:

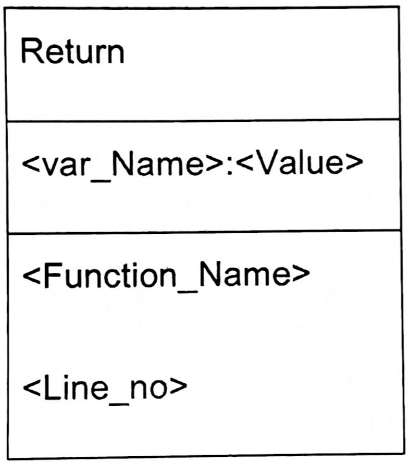
```
def fibonacci(n):
  a = 0
  b = 1
  if n < 0:
    print("Incorrect input")
  elif n == 0:
    return a
  elif n == 1:
    return b
  else:
    for i in range(2, n+1):
      c = a + b
      a = b
      b = c
    return b

print(fibonacci(9))
```

How Recursion work: Run time Stack

Runtime Stack: an area of memory set aside for programs / functions to use while they are running. The system stores local variables, function parameters and other system information in this area. The size of this area grows and shrinks during program execution as functions are called and then returned from.

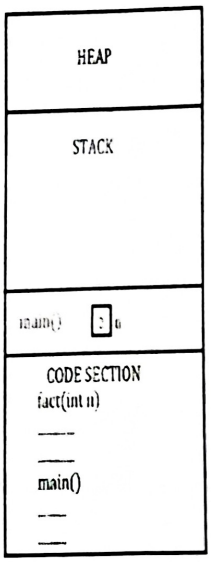
What is run time stack?

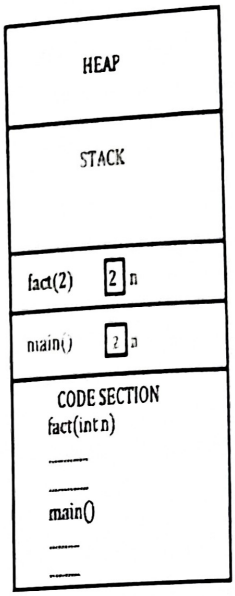
This is the simple run time stack in which call frames are added on to stack and they are popped one by one until the it reaches the last frame in the stack and then execution is terminated.




```
def fact(n):
    if n == 1:
        return 1
    return fact(n - 1)

def main():
    p = fact(2)

if __name__ == "__main__":
    main()
```

For the above program. Firstly, the activation record for main stack is generated and stored in the stack.

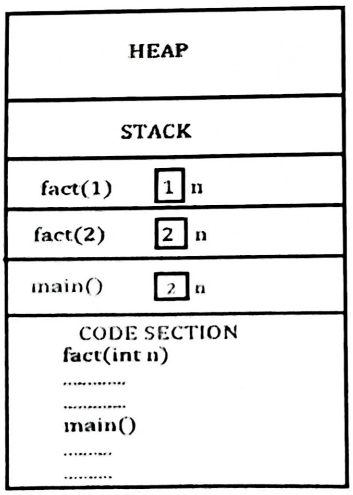




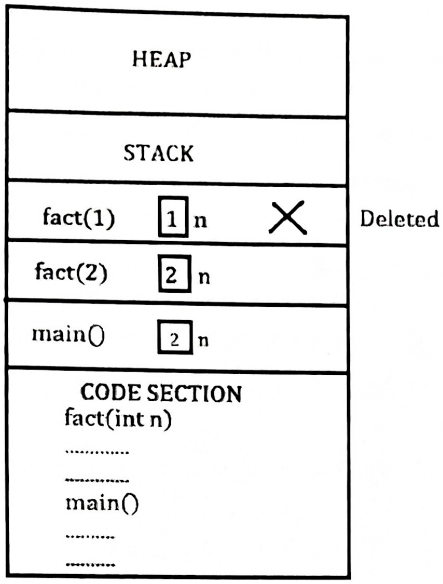
In the above program, there is a recursive function fact that has n as the local parameter. In the above example program, n=2 is passed in the recursive function call.

First Step: First, the function is invoked for n =2 and its activation record are created in the recursive stack.

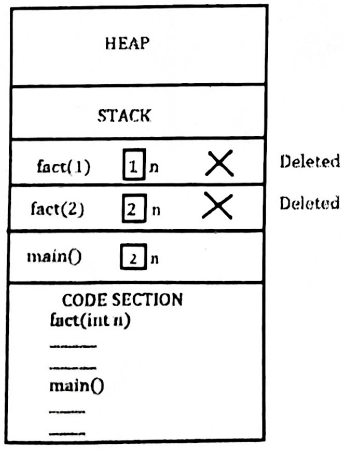
2nd Step: Then according to the recursive function, it is invoked for n=1 and its activation record is created in the recursive stack.



3rd Step: After the execution of the function for value n=1 as it is a base condition, its execution gets completed and its activation record gets deleted.



4th step: Similarly, the function for value n=2(its previous function) gets executed and its activation record gets deleted. It comes out from the recursive function to the main function.



Recursive Application:

1. **Recursive Binary search**

What is Recursive Binary Search

Define recursive binary search

Recursive algorithms are used in binary search. The broad strategy is to look at the middle item on the list. The procedure is either terminated (key found), the left half of the list is searched recursively, or the right half of the list is searched recursively, depending on the value of the middle element.

Example: Input: arr[] = {1, 4, 3, 5, 6, 8, 11, 10, 14, 17}
Target value = 8
OUTPUT: Element 8 is present at index 6.

```
def binarySearch(arr, left, right, number):

    if left > right:
            return -1

        mid = (left + right) // 2


        if number == arr[mid]:
            return mid


    elif number < arr[mid]:
            return binarySearch(arr, left, mid - 1, number)

        else:
            return binarySearch(arr, mid + 1, right, number)

arr = []
n = int(input('Enter the value of array size '))
for i in range(0,n):
    temp = int(input('Enter array value '))
    arr.append(temp)
number = int(input('Enter the target value '))

(left, right) = (0, len(arr) - 1)
index = binarySearch(arr, left, right, number)

if index != -1:
    print('Element found at index', index)
```

```
else:
    print('Element not found in the array')
```

2.Tower of Hanoi:

Tower of Hanoi is a mathematical puzzle where we have three rods (**A**, **B**, and **C**) and **N** disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod **A**. The objective of the puzzle is to move the entire stack to another rod (here considered **C**), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.
  - ***Input**: 2*
    ***Output:** Disk 1 moved from A to B*
    Disk 2 moved from A to C
    Disk 1 moved from B to C
  - ***Input:** 3*
    ***Output:** Disk 1 moved from A to C*
    Disk 2 moved from A to B
    Disk 1 moved from C to B
    Disk 3 moved from A to C
    Disk 1 moved from B to A
    Disk 2 moved from B to C
    Disk 1 moved from A to C

Tower of Hanoi using Recursion:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- *Shift 'N-1' disks from 'A' to 'B', using C.*
- *Shift last disk from 'A' to 'C'.*
- *Shift 'N-1' disks from 'B' to 'C', using A.*

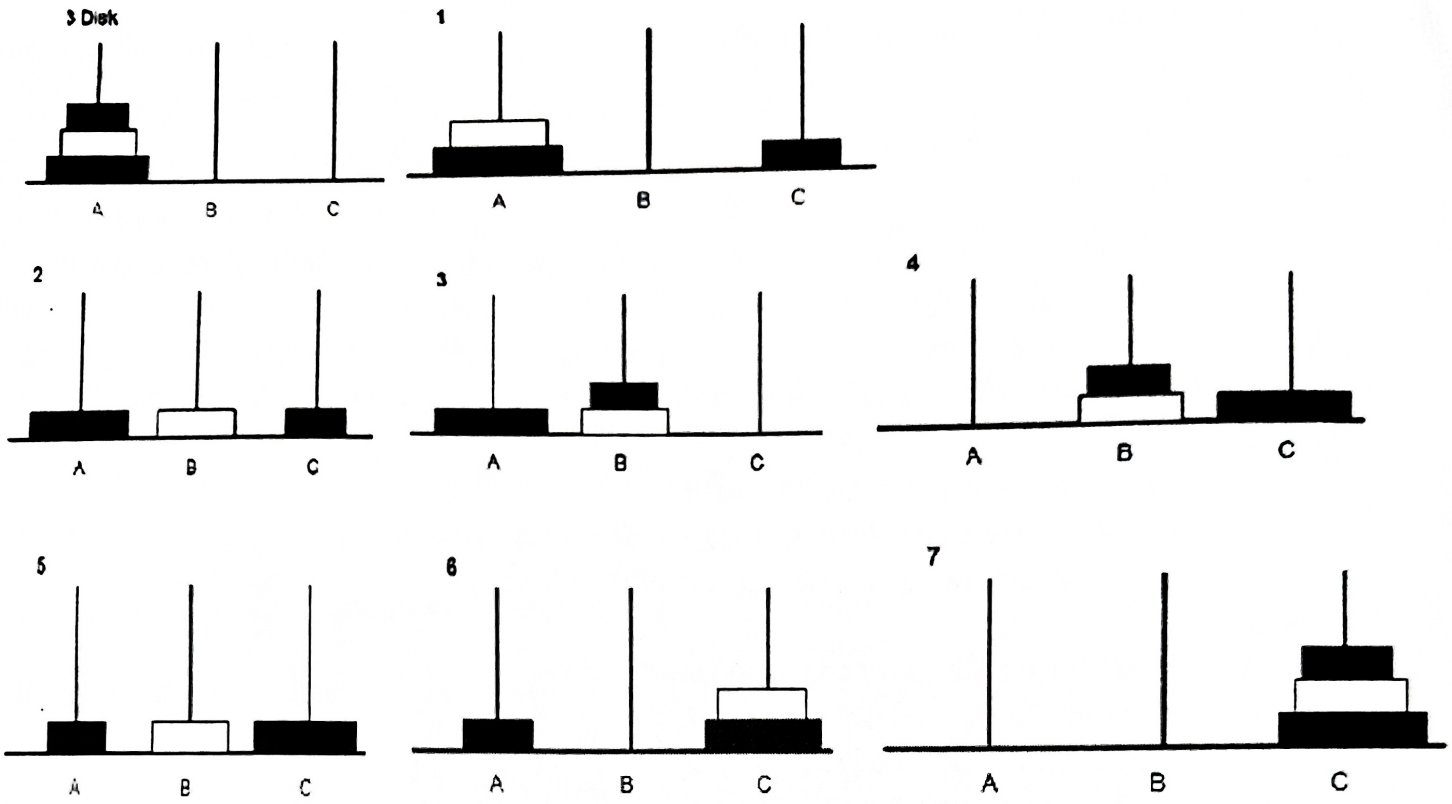


Image illustration for 3 disks