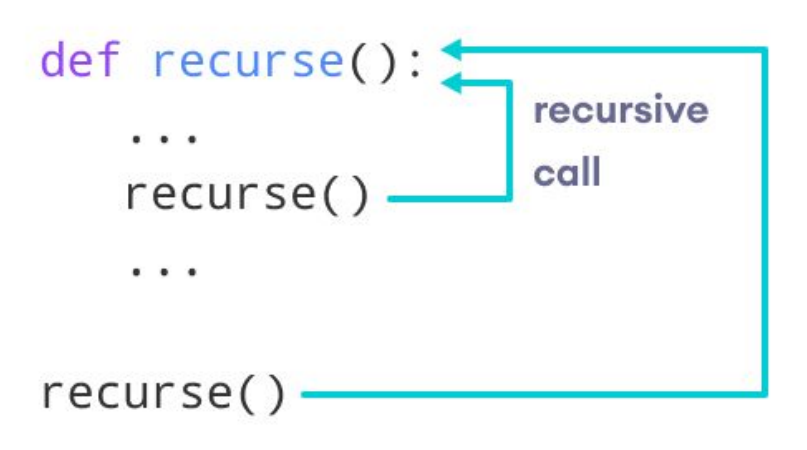
Unit 2: Recursion

Prepared By: Asst. Prof. Himani Desai

Contents

Introduction to Recursion

- There are two approaches to write the repetitive algorithms. One using iteration and another using recursion.
- Recursion is a repetitive process in which algorithm calls itself.
- The following image shows the working of a recursive function called recurse.



Factorial Algorithm using Recursion

Algorithm recursiveFactorial (n)

Calculates factorial of a number using recursion.

Pre n is the number being raised factorially

Post n! is returned

1 if (n equals 1)

1 return 1

2 else

1 return (n * recursiveFactorial (n - 1))

3 end if

end recursiveFactorial

Factorial Example using Recursion

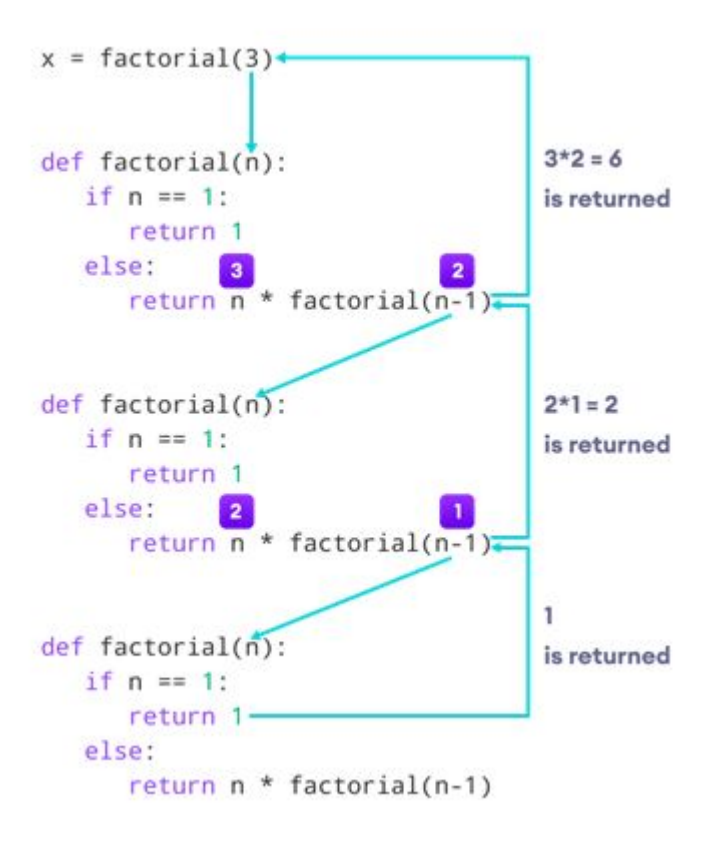
Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

Output

```
The factorial of 3 is 6
```

Factorial Example using Recursion (Cont.)



Time Complexity of Factorial

$$T(n) = T(n - 1) + 3$$
$$T(0) = 1$$

```
T(n) = T(n-1) + 3
     = T(n-2) + 6
     = T(n-3) + 9
     = T(n-4) + 12
     = ...
     = T(n-k) + 3k


as we know T(0) = 1
we need to find the value of k for which n - k = 0, k = n


T(n) = T(0) + 3n , k = n
     = 1 + 3n


that gives us a time complexity of O(n)
```

Designing Recursive Algorithms

- Hypothetically all recursive algorithms have two elements: each call either solves one part of the problem i.e., factorial(0) is 1.or reduces the size of problem by recursively calling factorial with n - 1.
- At each recursive call, the size of the problem is reduced, from the factorial of 3, to 2, and finally to 1.
- The statement that “solves” the problem is known as the base case. Every recursive algorithm must have a **base case**.
- The rest of the algorithm is known as the **general case.**
- The general case contains the logic needed to reduce the size of the problem.

Designing Recursive Algorithms (Cont.)

- Rules for designing a recursive algorithm:

1. First, determine the base case.

2. Then determine the general case.

3. Combine the base case and the general cases into an algorithm.

- In combining the base and the general case into an algorithm, we must pay careful attention to the logic.
- Each call must reduce the size of the problem and move it toward the base case. The base case, when reached, must terminate without a call to the recursive algorithm; that is, it must execute a return.

Limitation of Recursion

- Recursive solutions may involve extensive overhead (both time and memory) because they use calls. Each call takes time to execute.
- A recursive algorithm therefore generally runs more slowly than its nonrecursive implementation.
- Each time we make a call we use up some of our memory allocation. If the recursion is deep—that is, if there are many recursive calls—we may run out of memory.
- As a general rule, recursive algorithms should be used only when their efficiency is logarithmic.

Tower of Hanoi Puzzle

- Tower of Hanoi is a mathematical puzzle where we have three rods and n disks.
- The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules.

Rules of the game

The rules of **"Tower of Hanoi"** are quite simple, but the solution is slightly hard. There are three **rods**. The disks are stacked in the descending order; the largest disk stacked at the bottom and the smallest one on top.

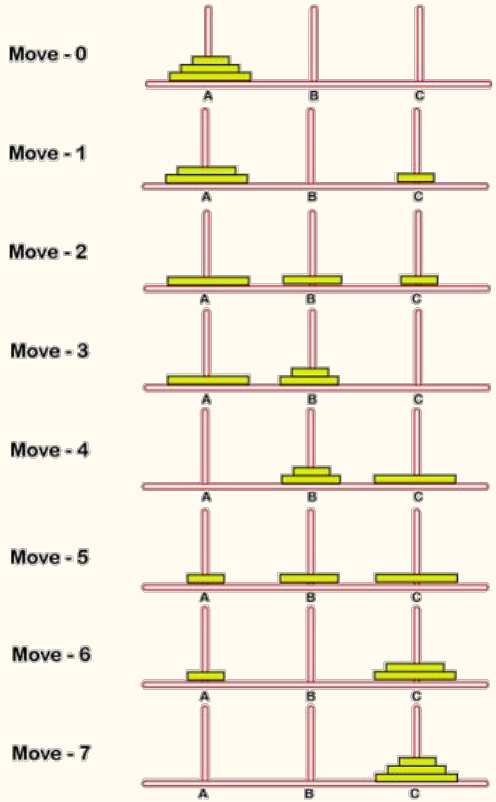
The task is to transfer the disks from one source rod to another target rod.

The following rule must not be violated

- Only one disk can be moved at a time.
- The most upper disk from one of the rod can be stimulated in move.
- The smaller disk cannot be placed at the lower of the largest disk.

The number of moves can be calculated as $\mathbf{2^n - 1}$.

Tower of Hanoi Puzzle Solution



Tower of Hanoi Puzzle (Cont.)

Problem Approach

1. Create a **tower_of_hanoi** recursive function and pass two arguments: the number of disks n and the name of the rods such as **source, aux,** and **target**.
2. We can define the base case when the number of disks is 1. In this case, simply move the one disk from the **source** to **target** and return.
3. Now, move remaining n-1 disks from **source** to **auxiliary** using the target as the **auxiliary**.
4. Then, the remaining 1 disk move on the **source** to **target**.
5. Move the n-1 disks on the auxiliary to the target using the source as the auxiliary.

Tower of Hanoi Puzzle (Cont.)

```
def tower_of_hanoi(disks, source, auxiliary, target):
    if(disks == 1):
        print('Move disk 1 from rod {} to rod {}.'.format(source, target))
        return
    # function call itself
    tower_of_hanoi(disks - 1, source, target, auxiliary)
    print('Move disk {} from rod {} to rod {}.'.format(disks, source, target))
    tower_of_hanoi(disks - 1, auxiliary, source, target)
disks = int(input('Enter the number of disks: '))
# We are referring source as A, auxiliary as B, and target as C
tower_of_hanoi(disks, 'A', 'B', 'C')  # Calling the function
```

Time Complexity of Fibonacci Series

The time complexity of the iterative code is linear, as the loop runs from 2 to n, i.e. it runs in O(n) time.

To calculate the time complexity of the code, we can solve a recurrence relation:

$T(0) = 0,$

$T(1) = 1,$

$T(n) = T(n-1) + T(n-2) + 4$

4 => The comparison and the two subtractions and one addition

Time Complexity of Fibonacci Series

We can approximate T(n-2) ~ T(n-1) as T(n-2) $\leq$ T(n-1)

```
T(n) = T(n-1) + T(n-2) + c
     = 2T(n-1) + c    //from the approximation T(n-1) ~ T(n-2)
     = 2*(2T(n-2) + c) + c
     = 4T(n-2) + 3c
     = 8T(n-3) + 7c
     = 2^k * T(n - k) + (2^k - 1)*c

Let's find the value of k for which: n - k = 0
k = n

T(n) = 2^n * T(0) + (2^n - 1)*c
     = 2^n * (1 + c) - c

i.e. T(n) ~ 2^n
```

Hence the time taken by recursive Fibonacci is O(2^n) or exponential.

Thank You