Function Recursion:

1. Recursion is the process of defining something in terms of itself.
2. When a function calls itself, it is known as recursion.
3. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

```
def recurse():
        ...
        recurse()

recurse()
```

Recursive call

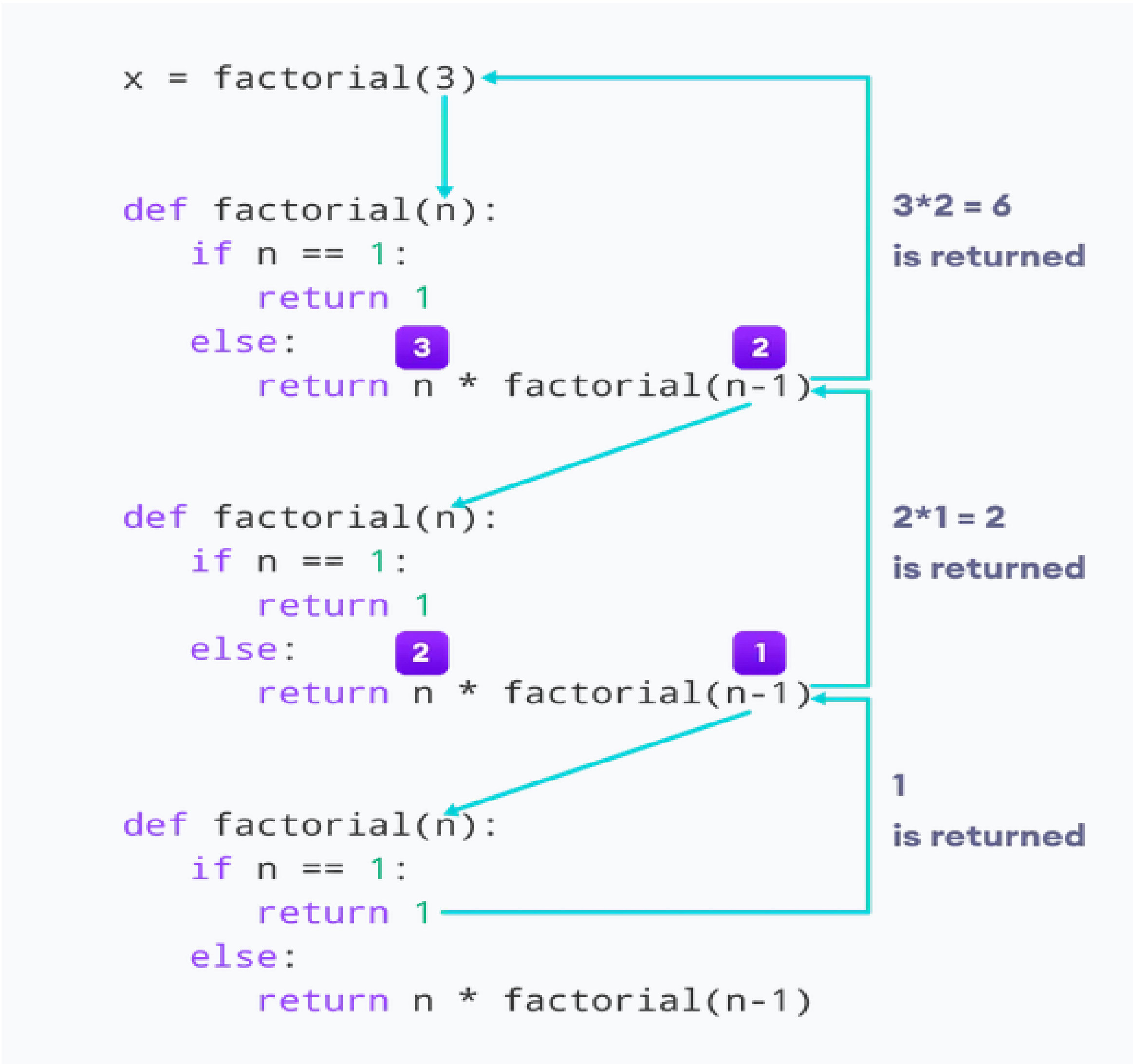
Example of recursive function (Program of Factorial):

```
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))
```

Recursive call:

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

Working:



Advantages:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Tail Recursion:

1. A unique type of recursion where the last procedure of a function is a recursive call.
2. The recursion may be automated away by performing the request in the current stack frame and returning the output instead of generating a new stack frame.
3. The tail-recursion may be optimized by the compiler which makes it better than non-tail recursive functions.

Q 1. Explain the step by step working of this code and predict the output:

```
def fun_power(n, p):
    if(p == 0 or p == 1):
        return n
    else:
        return (n* fun_power (n, p-1))
n = 3
p = 3
print(fun_power(n, p))
```

Sol.

27

Q2. Explain the step by step working of this code and predict the output

```
def fun_recursive(n):
    print("Calculating F", "(", n, ")", sep="", end=", ")

    # Base case
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Recursive case
    else:
        return fun_recursive (n-1) + fun_recursive(n-2)
fun_recursive(3)
```

Sol.

```
  Calculating F(3), Calculating F(2), Calculating F(1), Calculating F(0),
Calculating F(1),
```

2

Q3. Explain the step by step working of this code and predict the output:

```
sum = 0
def list1(lst):
    global sum
    for j in range(len(lst)):
        if type(lst[j]) == list:
            list1(lst[j])
        else:
            sum += lst[j]
list1((1,2,3,4,5,6,7))
print(sum)
```

Sol.

```
28
```

Q4. Explain the step by step working of this code and predict the output

```
def printPattern(targetNumber) :

  # Base Case
  if (targetNumber <= 0) :
    print(targetNumber, end = ' ')
    return

 # Recursive Case
  print(targetNumber, end = ' ')
  printPattern(targetNumber - 4)
  print(targetNumber, end = ' ')

# Driver Program
n = 8
printPattern(n)
```

Sol.

```
8 4 0 4 8
```

Q5. Explain the step by step working of this code and predict the output:

```
def fun2(n, x):
    if(n > 1):
        return (fun2 (n-1, x)+(n-1)* fun2(n-2, x))
    elif(n == 0):
        return x
    else:
        return  n
n = 4
X = 4
print(fun2(n, X))
```

Sol.

22

Q6. Explain the step by step working of this code and predict the output:

```
def fun1(array):
  if len(array) > 1:
    return [array[len(array) - 1]] + fun1(array[:len(array) - 1])
  elif len(array) == 1:
   return array
  else:
    return []
# Driver Code
array = [11, 12, 13, 14]
print(fun1(array))
```

```
Sol.
    [14, 13, 12, 11]
```

Q7. Think of a recursive version of the function f(n) = 3 * n,
i.e., the multiples of 3

Mathematically, we can write it like this:

f(1) = 3,
f(n+1) = f(n) +3,

Sol.

A Python function can be written like this

```
def mult3(n):
    if n == 1:
        return 3
    else:
        return mult3(n-1) + 3
for i in range(1,10):
        print(mult3(i))
```

Q8. Think of a recursive version of the factorial of a number

```
def recursive_factorial(n):
    if n == 1:
         return n
    else:
        return n * recursive_factorial(n-1)

# user input
num = 4

# check if the input is valid or not
if num < 0:
    print("Invalid input ! Please enter a positive number.")
elif num == 0:
    print("Factorial of number 0 is 1")
else:
    print("Factorial of number", num, "=", recursive_factorial(num))
```

Q9. Think of a recursive version to calculate the sum of the positive integers of n+(n-2)+(n-4)... (until n-x =< 0).

Sol.

```
def sum_series(n):
  if n < 1:
    return 0
  else:
    return n + sum_series(n - 2)

print(sum_series(6))
print(sum_series(10))
```

Q10. In mathematics, a geometric series is a series with a constant ratio between successive terms. We write a geometric sequence in this form

$$\{a, ar, ar^2, ar^3, \ldots ar^{n-1} \ldots\}$$

where *a* is the first item, and *r* is the common ratio between terms; ar^{n-1} represents the n^{th} item.

For example, let *a = 7* and *r = 2*. The following shows the first five items in the sequence:

$$\{7, 14, 28, 56, 112 \ldots\}$$

We can formalize the sums of the sequence as:

$$s_n = \begin{cases} a, & n = 1 \\ s_{n-1} + ar^{n-1}, & n > 1 \end{cases}$$

Think of a recursive version to calculate the sum geometric series (take n = 4, a = 7, and r = 2)

Sol.

```
def computeSum(n,a,r):
    if n == 1:
        return a
    else:
        # Make a recursive function call
        res = computeSum(n-1, a, r) + a*r**(n-1)
        return res
n = 5
a = 7
r = 2
sum = computeSum(n,a,r)
print('The sum of the first {} items in the sequence is {}.'.format(n,
sum))
```

output: The sum of the first 5 items in the sequence is 217.