6 Recursion

In This Chapter

6.1 Introduction

6.2 Recursive Function

6.3 How Recursion Works

6.4 Recursion in Python

6.5 Recursion *vs.* Iteration

6.1 INTRODUCTION

You have learnt how to create and invoke methods/functions in Python. Inside the function bodies, we can include function-calls of other functions. But have you ever thought : Could a method invoke or call itself ?

Self-invocation may at first sound useless or illegal : Isn't this defining something in terms of itself — what is called a circular definition? But self-invocation is legal, and it's actually quite useful. In fact, it's so useful that it gets its own special name : **recursion**.

This chapter explores recursion and recursive functions along with some classic recursive examples.

RECURSION

Recursion refers to a programming technique in which a function calls itself either directly or indirectly.

6.2 RECURSIVE FUNCTION

In many of your programs, you must have written some functions that **call** or **invoke** other functions, *e.g.*,

```
def A( ) {
    B ( )
```

Function A( ) is calling another function, B( )

Now, if you write a function definition that **calls itself**, then this function would be known as **recursive function**.

That is, a function is said to be recursive if the function definition includes a call to itself *e.g.*,

```
def A( ) :
    A( )
```

*Now function A( ) is called itself from its own function-body, so it is a **recursive function** now.*

See, another recursive function :

```
def recur( ) :
    recur( )            # calling itself
```

Recursion can be indirect also. The above example, where a function calls itself directly from within its body is an example of *direct recursion*. However, if a function calls another function which calls its caller function from within its body, it is known as *indirect recursion e.g.*,

```
def A( ) :
    B( )

def B( ) :
    A( )
```

Function A( ) calling B( ), which calls A( )

RECURSIVE FUNCTION

A function is said to be **Recursive Function** if it calls itself.

So, you can say that recursion can be :

- **direct recursion**, if a function calls itself directly from its function body.
- **indirect recursion**, if a function calls another function, which calls its caller function.

Following figure illustrates it.

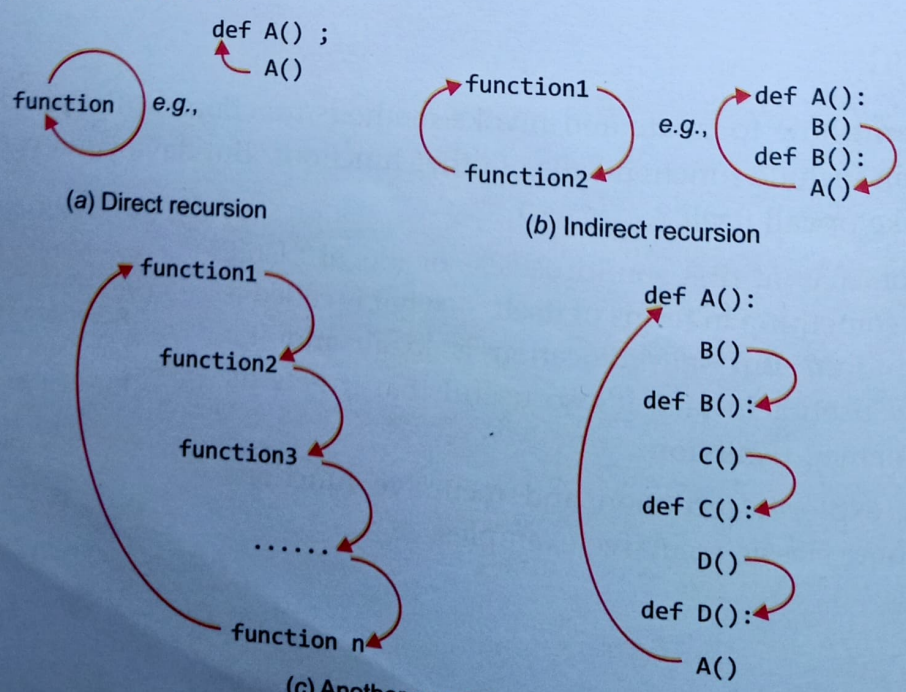


Figure 6.1 Types of Recursion.

The above shown sample code is an example of *indirect recursion*.

In mathematics also, you have seen recursive problems *e.g.*, sum of n natural numbers can be written as

sum of n natural numbers = n + sum of $n-1$ natural numbers

Similarly, sum of $n-1$ natural numbers = $n-1$ + sum of $n-2$ natural numbers

Similarly, sum of $n-2$ natural numbers = $n-2$ + sum of $n-3$ natural numbers,

and so on.

This section is going to discuss recursion and recursive functions. We shall be learning how to write recursive functions and implement recursion. But before we proceed, carefully go through the following code and figure out its output.

```
def func1( )
    print("Hello func2")
    func2( )

def func2( )
    print("Yes func1")
    func1( )
```

Yes, you guessed it right. This code will print the following ENDLESSLY :

```
Hello func2
Yes func1
Hello func2
Yes func1
    ⋮
```

The above functions (*func1( )* and *func2( )*) will keep on calling one another infinitely and the entire memory will be used up – No stop!

Python will report following runtime error for it :

```
RuntimeError: maximum recursion depth[1] exceeded . . .
```

The problem with above code is that it is a Recursive program but recursion code is NOT SENSIBLE. You must write a sensible recursive code that has clearly defined 'where to stop'. Let us see how.

RECURSION

Recursion is a technique for solving a large computational problem by repeatedly applying the same procedure(s) to reduce it to successively smaller problems. A recursive procedure has *two* parts: one or more base cases and a recursive step.

Right or Sensible Recursive code is the one that fulfills following requirements :

- it must have a case, whose result is known or computed without any recursive calling – the **BASE CASE. The BASE CASE must be reachable for some argument/parameter.**
- it also have **Recursive Case**, where by function calls itself later section, (Section 6.4, explains this clearly).

1. Recursion limit is platform dependent and can be set via *sys.setrecursionlimit(limit)*.

To understand the above points, let us consider an example program first and see how it works.

Program 6.1 Write a recursive function that computes the sum of numbers 1..n ; get the value of last number *n* from user.

```
1.  # program to illustrate recursion
2.
3.  def compute (num):
4.      if (num == 1) :
5.          return 1
6.      else:
7.          return (num + compute (num - 1) )
8.
9.  # __main__
10. last = 4
11. ssum = compute (last)
12. print ("The sum of the series from 1 ..", last, "is", ssum)
```

*For **num's** value 1, the result is pre-known and is available (or calculated) without any recursive call*

***compute( )** calling itself – recursive code*

*Initial call to function **compute( )***

The output produced by above code is :

```
The sum of the series from 1 .. 4 is 10
```

Let us now understand how it (program 6.1) works. Lines 3-7 of program 6.1 define a function **compute( )**, which is a recursive function as it calls itself on *line 7*.

1. Program starts __main__ part at *line 9* and then calls **compute( )** with variable **last** that has value 4, *i.e.*, **compute(4)** is invoked at line 11 initially.

2. With function call compute(4) at *line 11*, control shifts to *line 3*, where the code of **compute( )** begins ; parameter **num** takes value 4.

   2.1 At *line 4*, condition **num == 1** is *false*, so control shifts to else part (*line 6-7*)

   2.2 *Line 7* calculates *return value* as **num + compute(num-1)**, *i.e.*, as 4 = computer (4-1), *i.e.*, **4+compute(3)**.

   Since the value of **compute(3)** is not known, function compute( ) is to be called with value 3.

   2.2.1 So compute( ) is again called with value 3 so **num** takes value **3**. In line 4, *num* (which is 3) is still not 1 (condition 3 == 1 is *false*,) so line 7 (its return value) gets computed as **3 + compute (3-1)** *i.e.*, **3 + compute(2)**.

   2.2.1.1A For **compute(2)**, line 4 condition **num == 1** is false (as num is 2) so line 7 (its return value) is executed as **2 + compute(1)**

   2.2.1.1B For **compute(1)** condition of line 4 (num == 1) is true as **num is 1** now. So **compute (1)** returns value 1 as line 5 gets executed and control returns to **2+compute(1)** (2.2.1.1A) with value 1 for **compute (1)**.

2.2.1.1-R So after returning to 2.2.1.1.A, the compute (2)'s return value is calculated as **2 + compute(1)** = 2 + 1 = **3** (value of **compute(1)** is 1). It calculates the **compute(2)'s** return value as **3**. So the control returns to 2.2.1.

2.2.1.R **compute(2)** returns value 3. So return value of 3 gets computed as **3+compute(2)** = **3 + 3 = 6**. Now the control returns to 2.2.

2.2R **compute(3)** returns 6 and thus the return value of **compute (4)** gets computed as 4 + 6 = **10** and control returns to line **11** that called **compute(4)** as **ssum = compute(4)**.

3. Now variable **ssum** gets the return value of **compute(4)** as 10 and then at line 12, it prints the *sum*.

What this program manages to do is to display the value of 4 + (3 + (2 + 1)). That is, it displays the sum of the numbers from 4 down to 1. More generally, what this *compute* method does is to return the sum of all the integers between 1 and its parameter *n*.

6.3 HOW RECURSION WORKS

You'll notice that the *compute* function doesn't *always* recur: when its parameter *n* is 1, the function simply returns immediately without any recursive invocations. (*line 5*)

With a bit of thought, you'll realize that any functional recursive function must have such a situation, since otherwise, the recursive function will never finish. In fact, these situations are important enough to designate special case(s) where result is pre-known without invoking the function again : Any condition where a recursive function does not invoke itself is called a **base case.**

But what exactly happens when a recursive function lacks a base case ? To understand this, we need to get some idea about how a computer handles function invocations.

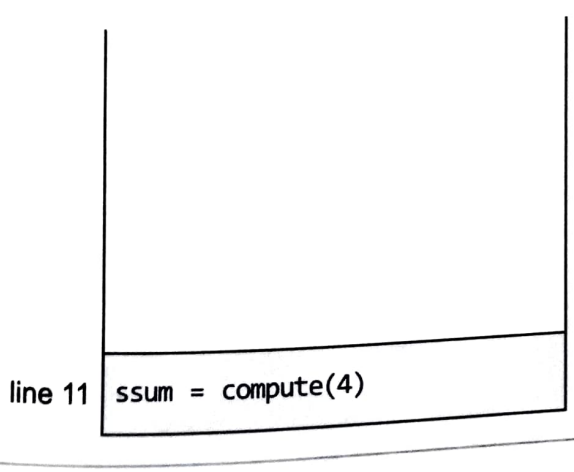
In executing a program, the computer creates what is called the **function stack**. The function stack is a stack of **frames**, each frame corresponding to a function invocation. At all times, the computer works on executing whichever function is at the stack's top; but when there is a function invocation, the computer creates a new frame and places it atop the stack. When the function at the stack's top returns, the computer removes that function's frame from the stack's top, and resumes its work on the function now on the frame's top.

To see how this works, let's diagram how the program 6.1 operated.

To see recursion in action

Scan QR Code

1.



To start off the program, the program pushes a frame corresponding to an invocation to. Notice how this frame includes room for return value of compute(4) in variable ssum.

2.

	compute(4)
line 7	num = 4, return 4 + compute(3)
line 11	ssum = compute(4)

When the computer invokes *compute*(4), the computer places a new frame atop the stack corresponding to *compute* ; this frame will include the variable *num*, whose value is initially 4.

3.

	compute(3)
line 7	num = 3, return 3 + compute(2)
	compute(4)
line 7	num = 4, return 4 + compute(3)
line 11	ssum = compute(4)

When the computer sees that *compute*(4) invokes *compute*(3), the computer places a new frame atop the stack, containing the variable *num*, whose value is initially 3.

4.

	compute(2)
line 7	num = 2, return 2 + compute(1)
	compute(3)
line 7	num = 3, return 3 + compute(2)
	compute(4)
line 7	num = 4, return 4 + compute(3)
line 11	ssum = compute(4)

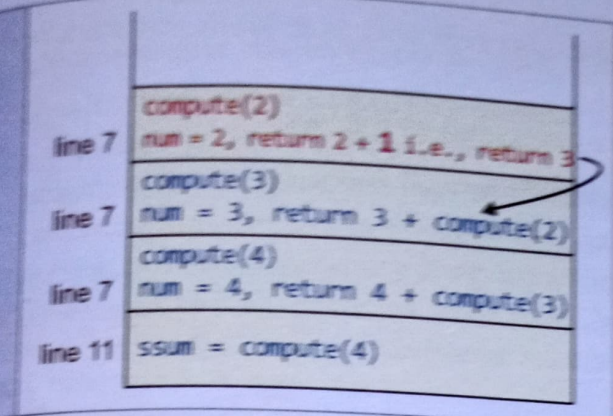
When the computer sees that *compute*(3) invokes *compute*(2), the computer places a new frame atop the stack, containing the variable *num*, whose value is initially 2.

5.

	compute(1)
line 7	num = 1, return 1 — line 5
	compute(2)
line 7	num = 2, return 2 + compute(1)
	compute(3)
line 7	num = 3, return 3 + compute(2)
	compute(4)
line 7	num = 4, return 4 + compute(3)
line 11	ssum = compute(4)

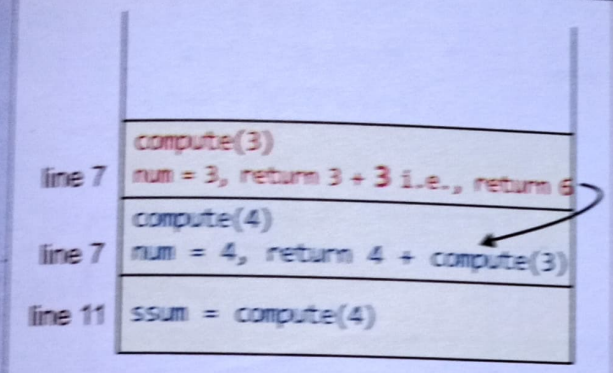
When the computer sees that *compute*(2) invokes *compute*(1), the computer places a new frame atop the stack, containing the variable *n*, whose value is initially 1.

6.



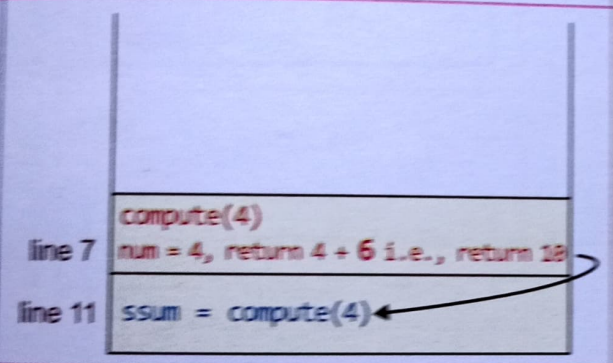
When the computer sees that *compute*(1) returns, it pops the top frame off the stack and resumes with whatever frame is now at the top – which happens to be the frame for *compute*(2). Value for compute (1) is replaced with its return value which is 1

7.



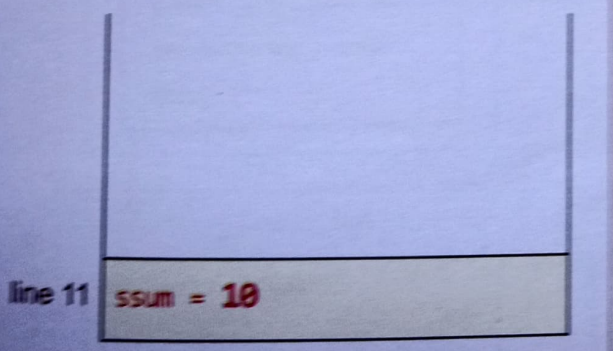
When the computer sees that *compute*(2) returns with value3, it pops the top frame off the stack and resumes with *compute*(3).

8.



When the computer sees that *compute*(3) returns with value 6, it pops the top frame off the stack and resumes with *compute*(4).

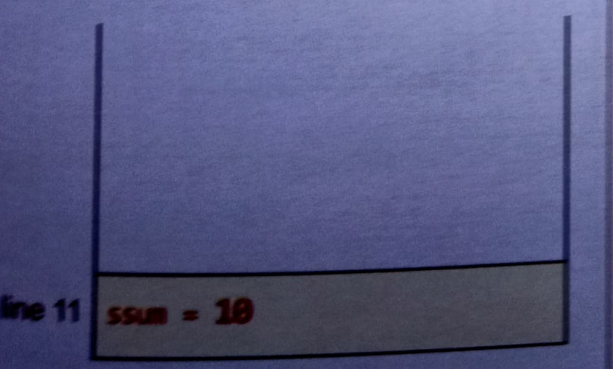
9.



When the computer sees that *compute*(4) returns with value 10, it pops the top frame off the stack and resumes with statement.

`ssum = compute(4) i.e., ssum = 10`

10.



Now the last line of program is executed and result is printed as

The sum of series from 1..4 is 10

And after this, the program ends and its frame is also removed from stack and stack becomes empty.

So we can say that above given compute(4) was executed as follows :

compute (4)

= 4 + compute (3)

= 4 + (3 + compute (2))

= 4 + (3 + (2 + compute(1)))

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

= 4 + (3 + 3)

= 4 + 6

= 10

NOTE

It is mandatory to have a base case in order to write a sensible recursion code.

Program 6.2 Write a recursive function to print a string backwards.

```
1.  # simple recursion
2.  def bp(sg, n):
3.      if n > 0:
4.          print(sg[n], end = '')
5.          bp(sg, n-1)  <------------  Recursive call to function bp( )
6.      elif n == 0 :
7.          print(sg[0])
8.
9.  # __main__
10. s = input("Enter a string : ")
11. bp(s, len(s)-1)
```

The output produced by above program is as shown below :

```
Enter a string : NEWS
SWEN
```

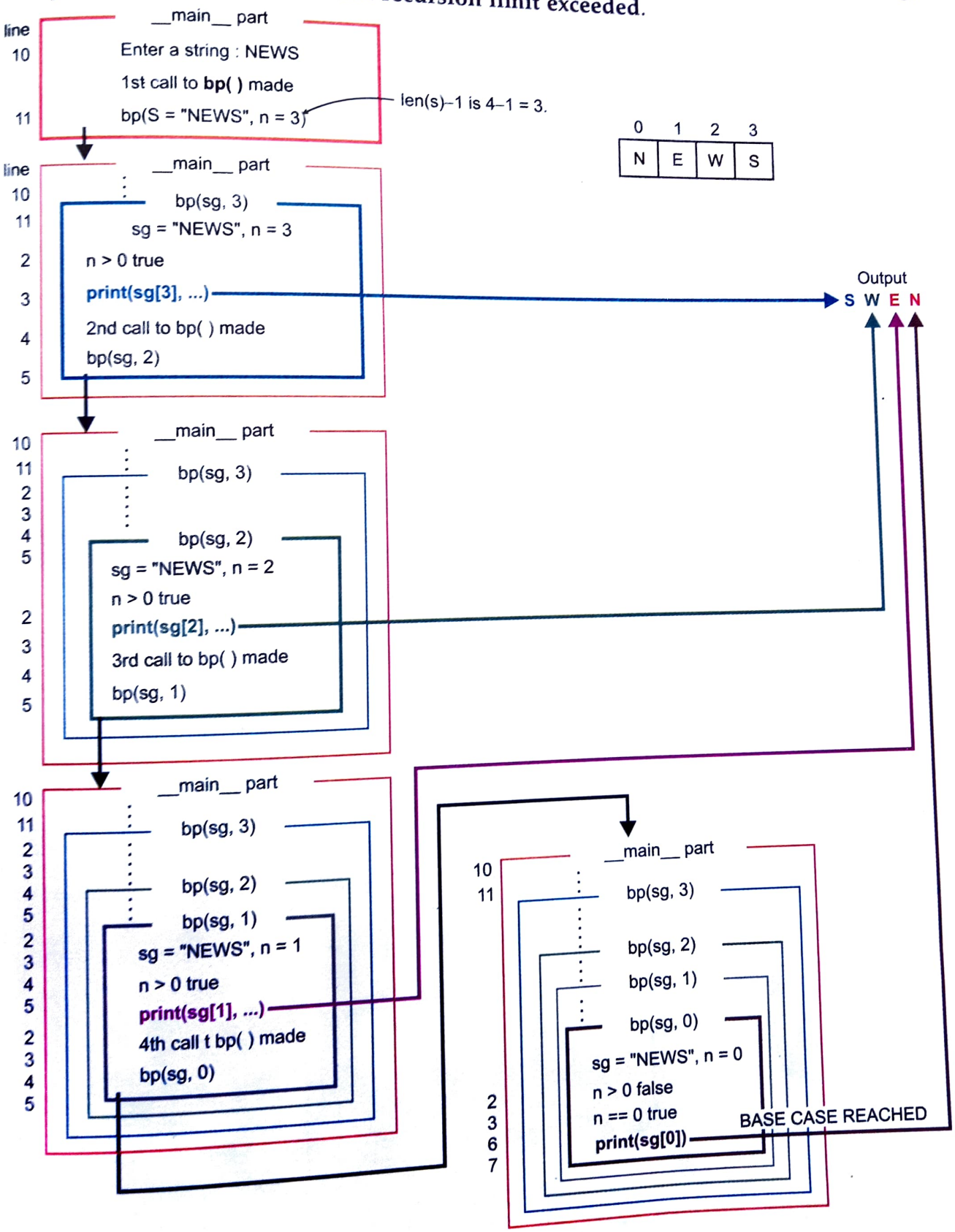
Can you figure out the flow of execution of above program? It is :

2 → 10 → 11 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 5 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 7

bp(sg = "NEWS", n = 3) | bp(sg = "NEWS", n = 2) | bp(sg = "NEWS", n = 1) | bp(sg = "NEWS", n = 0) Base Case

Let us see how this recursive program executed itself.

So **what happens if a recursive method never reaches a base case?** Hmm, you guessed it right – the stack will never stop growing. The computer, however, limits the stack to a particular height, so that no program eats up too much memory. If a program's stack exceeds this size, the computer initiates an exception, which typically would crash the program. (From the operating system's point of view, crashing the program is preferable to allowing a program to eat up too much memory and interfere with other better-behaved programs that may be running.) The exception is labelled as **maximum recursion limit exceeded**.



Consider another recursive function that also does the same thing as program 6.2 *i.e.*, backward printing of a string.

Program 6.3 Write a recursive function to print a string backwards.

```
# simple recursion
def bp (sg, n):
    if n > 0:
        k = len(sg) - n
        bp (sg, n-1)
        print(sg[k], end = ' ')
    elif n == 0 :
        return

# __main__
s = input("Enter a string : ")
bp(s, len(s))
```

Recursive call to function ***bp( )***

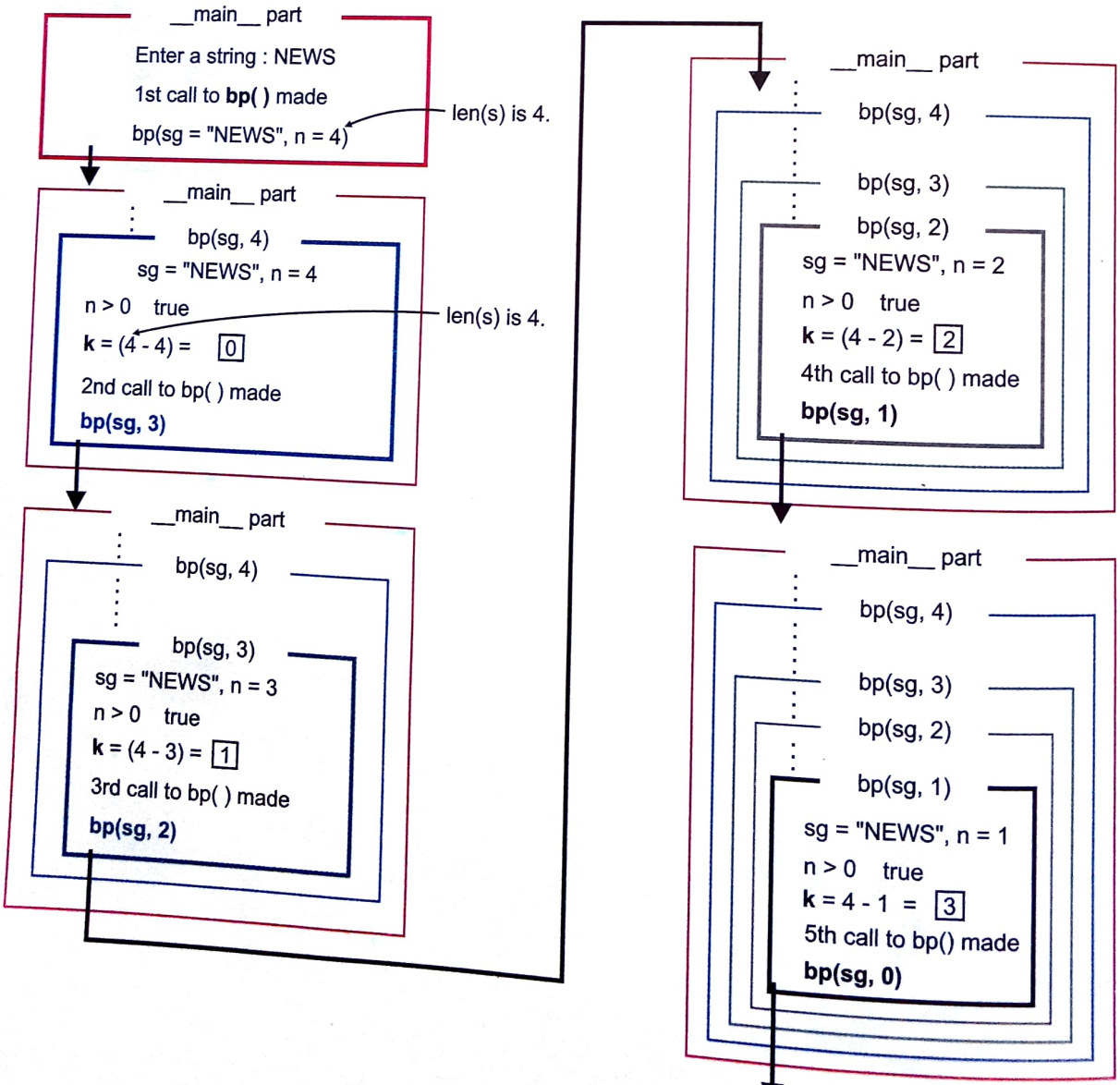
The output produced by above program is same as previous program, i.e. :

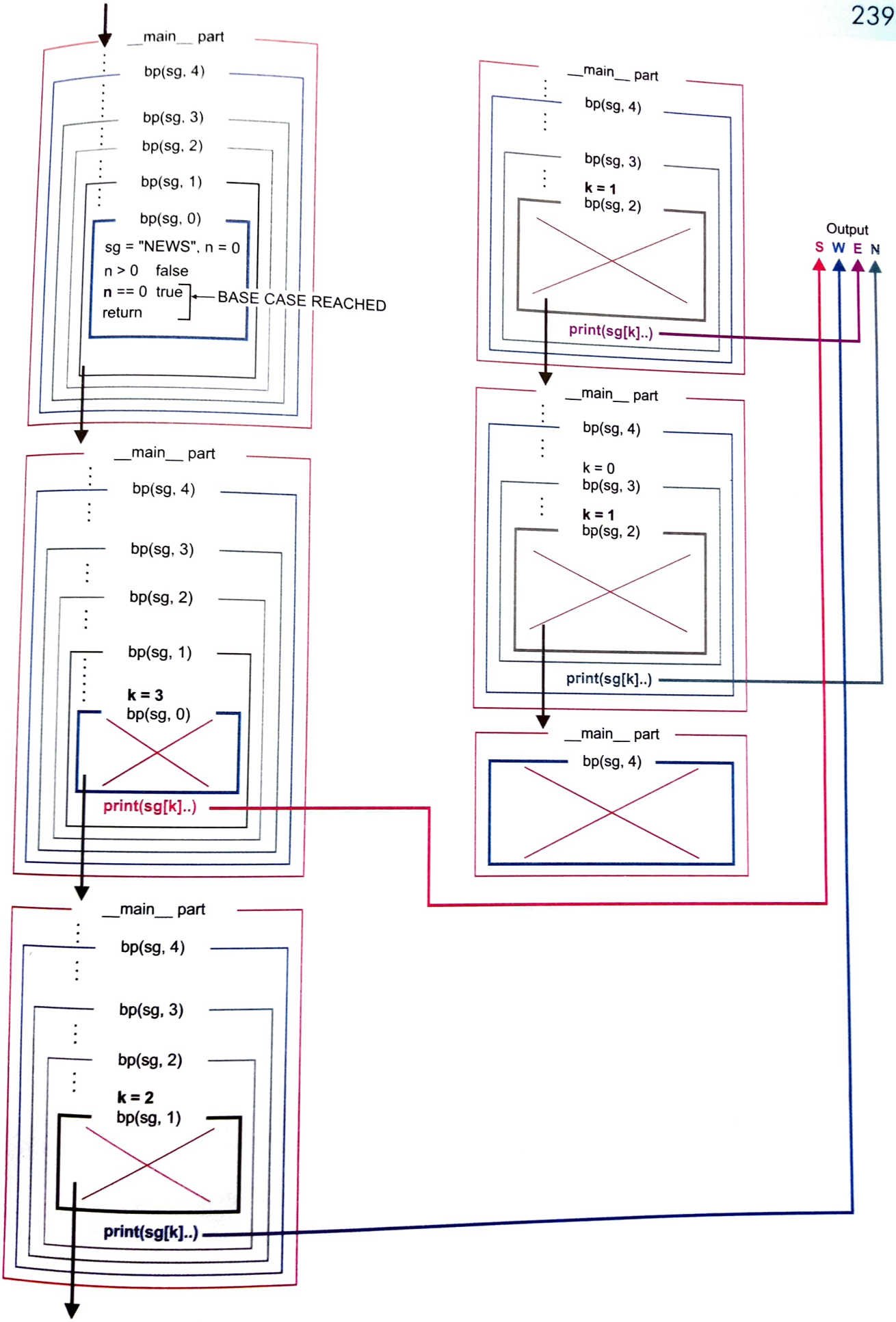
```
Enter a string : NEWS
SWEN
```

To see recursion in action

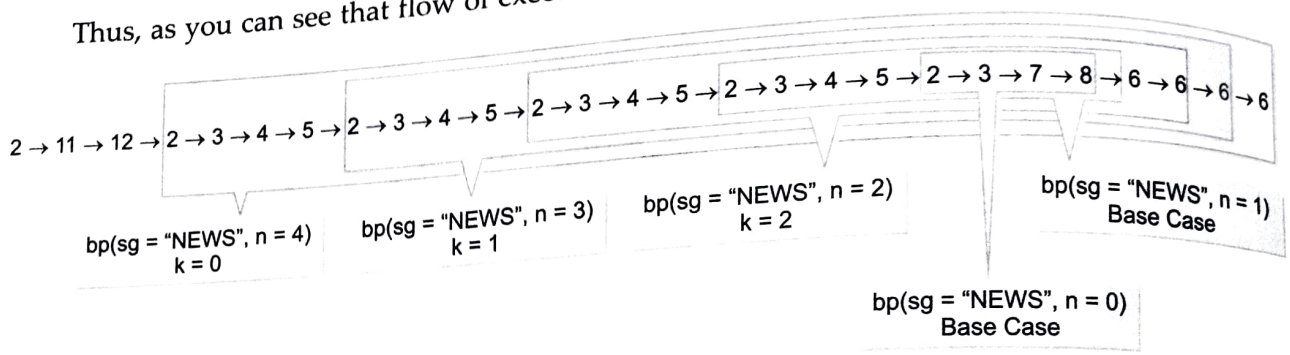
Scan QR Code

Let us see how above recursive program executed itself. Although it does the same thing as previous program 6.2, yet it is different from program 6.2.




__main__ part
bp(sg, 4)
bp(sg, 3)
bp(sg, 2)
bp(sg, 1)
bp(sg, 0)
sg = "NEWS", n = 0
n > 0 false
n == 0 true
return
BASE CASE REACHED
__main__ part
bp(sg, 4)
bp(sg, 3)
bp(sg, 2)
bp(sg, 1)
k = 3
bp(sg, 0)
print(sg[k]..)
__main__ part
bp(sg, 4)
bp(sg, 3)
bp(sg, 2)
k = 2
bp(sg, 1)
print(sg[k]..)
__main__ part
bp(sg, 4)
bp(sg, 3)
k = 1
bp(sg, 2)
print(sg[k]..)
__main__ part
bp(sg, 4)
k = 0
bp(sg, 3)
k = 1
bp(sg, 2)
print(sg[k]..)
__main__ part
bp(sg, 4)
Output
S W E N

Thus, as you can see that flow of execution of above program is as :



Technically, Python arranges the memory spaces needed for each function call in a *stack*. The memory area for each new call is placed on the top of the stack, and then taken off again when the execution of the call is completed. So, you can say that, the stack goes through the following sequence in recursive calls :

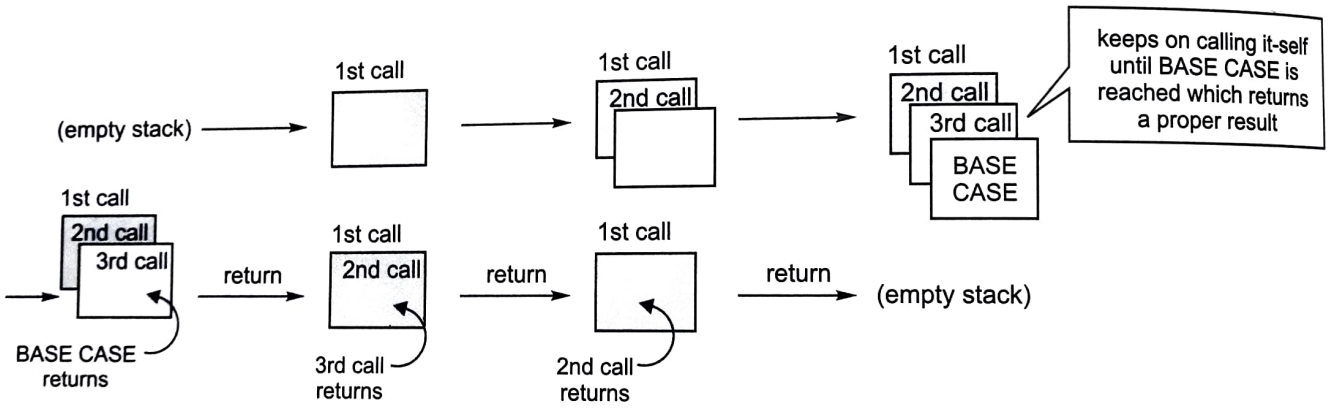


Figure 6.2 Memory manipulation in recursion.

Python uses this stacking principle for all nested function calls — not just for recursively-defined functions. A stack is an example of a "last in/first out" structure.

6.4 RECURSION IN PYTHON

As you have seen, recursion occurs when a function calls itself. In Python, as in other programming languages that support it, recursion is used as a *form of repetition that does not involve iteration.*

A **Recursive Definition** is a definition that is made in terms of a smaller version of itself. Have a look at following definition of x^n, which is non-recursive :

$$x^n = x * x * \ldots * x$$

(Iterative definition – **nonrecursive**)

Now, it can be represented in terms of recursive definition as follows :

$$x^n = x * (x^{n-1}) \quad \text{for } n > 1$$
$$= x \quad \text{for } n = 1$$
$$= 1 \quad \text{for } n = 0$$

(Recursive definition)

Writing a Recursive Function. Before you start working recursive functions, you must know that every recursive function must have at least *two* cases :

(*i*) the **Recursive Case** (or the **inductive case**)

(*ii*) the **Base Case** (or the **stopping case**) – ALWAYS REQUIRED

- The **Base Case** is a small problem that we know how to solve and is the case that causes the recursion to end. In other words, it is the case whose solution is pre-known (either as a value or formula) and used directly.
- The **Recursive Case** is the more general case of the problem we're trying to solve using recursive call to same function.

NOTE

The Base Case in a recursive program MUST BE REACHABLE.

As an example, with the power function x^n, the **recursive case** would be :

```
Power (x, n) = x * Power (x, n - 1)
```

and the **base cases** would be

```
Power (x, n) = x    when n = 1
```

and

```
Power (x, n) = 1    when n = 0
```

NOTE

Every recursive function consists of one or more base cases and a general, recursive case.

Other cases (when $n < 0$) we are ignoring for now for simplicity sake.

Consider the following example (program 6.4) of Power Function :

Program 6.4 *Program to show the use of recursion in calculation of power e.g., a^b*

```
# power a to b using recursion 6_4
def power(a, b) :
    if b == 0 :            # BASE CASE
        return 1
    else :
        return a * power(a, b-1)

# __main__
print("Enter only the positive numbers below")
num = int(input("Enter base number :"))
p = int(input("raised to the power of :"))
result = power(num , p)
print(num, "raised to the power of", p, "is", result)
```

The output produced by above program is as :

```
Enter only the positive numbers below
Enter base number : 7
raised to the power of : 3
7 raised to the power of 3 is 343
```

If *there is no base case, or if the base case is never executed, an* ***infinite recursion occurs***. An Infinite Recursion is when a recursive function calls itself endlessly. Infinite recursion can happen in one of the following cases.

(*i*) **Base Case Not Defined.** When the recursive function has no BASE CASE defined, *infinite recursion* will occur.

(*ii*) **Base Case Not reached.** Sometimes you have written a base case but the condition to execute it, is never satisfied, hence the **infinite recursion** occur.

For example, the code of above program 6.4 will face infinite recursion if you enter a negative value for power. In that case, the condition for base case, *i.e.*,

```
if b == 0 :
    return 1
```

This condition will never be true for a negative value of b

will never be satisfied and hence infinite recursion will occur.

To avoid infinite recursion, your code should take care of possible values that may cause infinite recursion, *e.g.*, above program (6.4)'s code can be modified as follows to avoid infinite recursion :

```
def power(a, b) :
    if b <= 0 :
        return 1
    else :
        return a * power(a, b-1)
# __main__
:
```

Now it takes care of negative values of b

NOTE

There can be one or more base cases in a recursive code.

NOTE

An Infinite Recursion is when a recursive function calls itself endlessly.

Or

You may check for negative value of power in *__main__ part* as well, *i.e.*, as :

```
def power(a, b) :
    :
# __main__
num = int(input("Enter base number :"))
p = int(input("raised to the power of :"))
if p < 0 :
    print ("Sorry, negative power not allowed")
else :
    result = power(num , p)
    print(num, "raised to the power of", p, "is", result)
```

Now it takes care of negative values of power

You can easily determine the flow of execution in above program.

Iterative Version

A recursive code may also be written in non-recursive way, which is the iterative version of the same. For instance, the iterative version of above program(6.4)'s code is given in program 6.5. below :

Program 6.5 Program to calculate a^b using iterative code.

```
# power a to b using iteration

def power(a, b) :
    res = 1
    if b == 0 :
        return 1
    else :
        for i in range(b):
            res = res * a
    return res

#__ main__
print("Enter only the positive numbers below")
num = int(input("Enter base number :"))
p = int(input("raised to power of :"))
result = power(num , p)
print(num, "raised to power of", p, "is", result)
```

Iterative version to calculate a^b.

The output produced by above program is as :

```
Enter base number : 3
raised to power of : 4
3 raised to power of 4 is 81
```

6.4.1 Some Recursive Codes

Let us write some Python codes that use recursion carrying out its task.

A. Computing Factorial Recursively

Now, consider another example. The factorial of a non-negative number is defined as the product of all the values from 1 to the number :

$$n! = 1 * 2 * \ldots * n$$

It can also be defined recursively as

$$n! = 1 \qquad \text{if } n < 2$$
$$= n * (n-1)! \qquad \text{if } n >= 2$$

Recursively factorial function would be written as follows :

```
# recursive factorial function 6.6
def factorial (n) :
    if n < 2:
        return 1
    return n * factorial(n-1)
```

```
# __main__
n = int(input("Enter a number (> 0) : "))
print("Factorial of", n, "is", factorial(n))
```

Sample runs of above program code are :

```
Enter a number (> 0) : 4
Factorial of 4 is 24
=========================
Enter a number (> 0) : 5
Factorial of 5 is 120
```

The iterative version of above given factorial function is (you already have written this) :

```
def factorial (n) :
    fact = n
    for I in range(1 , n):
        fact = fact * I
    return fact
```

B. Computing GCD recursively

We can efficiently compute the gcd using the following property, which holds for positive integers p and q :

If p > q

the gcd of p and q is the same as the gcd of p and p % q. That is, our Python code to compute GCD recursively would be :

```
def gcd (p, q) :
    if q == 0 :
        return p
    return gcd(q, p % q)
```

The **base case** is *when q is 0, with gcd(p, 0)* = p. To see that the reduction step converges to the base case, observe that the second input strictly decreases in each recursive call since p % q < q. If p < q the first recursive call switches the arguments.

```
gcd(1440, 408)
    gcd(408, 216)
        gcd(216, 24)
            gcd(192, 24)
                gcd(24, 0)
                return 24
            return 24
        return 24
    return 24
return 24
```

This recursive solution to the problem of computing the greatest common divisor is known as *Euclid's algorithm* and is one of the oldest known algorithms – it is over 2000 years old.

You can write recursive code for GCD computation yourself. (also refer to solved problem 15)

C. Fibonacci Numbers

You have written program to compute and display Fibonacci series using a loop. Here we are going to develop a recursive method to compute numbers in the Fibonacci sequence. This infinite sequence starts with 0 and 1, which we'll think of as the zeroth and first Fibonacci numbers, and each succeeding number is the sum of the two preceding *Fibonacci numbers*. Thus, the ***third term*** is 0 + 1 = 1. And to get the ***fourth term*** Fibonacci number, we'd sum the *2nd term* (1) and the *3rd term* (1) to get 2. And the ***fifth term*** is the sum of the *3rd term* (1) and the *4th term* (2), which is 3. And so on.

n	:	0	1	2	3	4	5	6	7	8	9	10	11	...
nth Fibonacci	:	0	1	1	2	3	5	8	13	21	34	55	89	...

We want to write a method *fib* that takes some integer *n* as a parameter and returns the *n*th Fibonacci number, where we think of the first 1 as the first Fibonacci number. Thus, an invocation of *fib*(6) should return 8, and the invocation of *fib*(7) should return 13.

```
def fib(n) :
    if n == 1 :                 # 1st term is 0
        return 0
    elif n == 2 :               # 2nd term is 1
        return 1
    else :
        return fib(n -1) + fib(n -2)
```

These are the BASE CASES – value is returned without invoking the function

RECURSIVE CASE – the fib( ) function is invoking itself from its own body

In talking about recursive procedures such as this, it's useful to be able to diagram the various method calls performed. We'll do this using a recursion tree. The recursion tree for computing *fib*(5) is in Fig. 6.3.

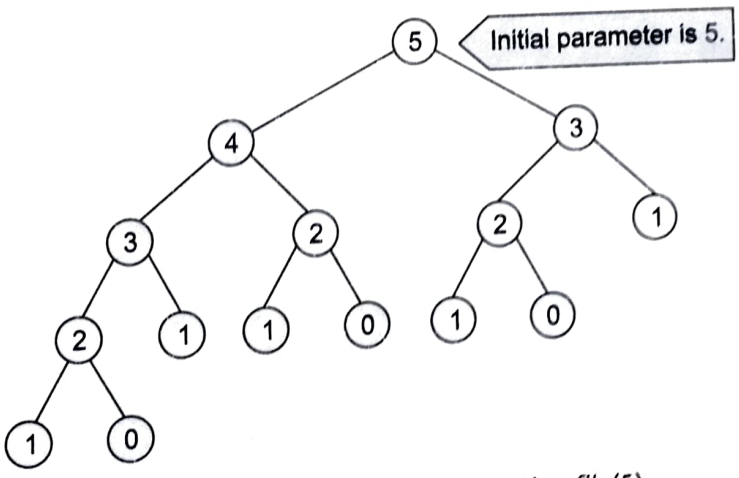


Figure 6.3 Recursion tree for computing *fib* (5).

The recursion tree has the original parameter (5 in this case) at the top, representing the original method invocation. In the case of *fib*(5), there would be two recursive calls, to *fib*(4) and *fib*(3), so

we include 4 and 3 in our diagram below 5 and draw a line connecting them. Of course, *fib*(4) has two recursive calls itself, diagrammed in the recursion tree, as does *fib*(3). The complete diagram in Fig. 6.3 depicts all the recursive invocation of *fib* made in the course of computing *fib*(5). The bottom of the recursion tree depicts those cases when there are no recursive calls — in this case, when $n <= 1$.

Following program lists the complete code of using recursive fib( ) function given above.

Program 6.6 Program using a recursive function to print fibonacci series upto nth term.

```
def fib(n) :
    if n == 1 :                          # 1st term is 0
        return 0
    elif n == 2 :                        # 2nd term is 1
        return 1
    else :
        return fib( n -1 ) + fib( n -2 )
# __main__
n = int(input("Enter last term required :"))
for i in range(1, n + 1 ) :              # list with values 1..n
    print(fib(i), end = ', ')
print("...")
```

The output produced by above program is :

```
Enter last term required : 8
0, 1, 1, 2, 3, 5, 8, 13, ...
```

6.4.2 Binary Search

Another popular algorithm that uses recursion successfully is **binary search** algorithm. But hey, we have not talked about binary search before. No worries! We shall first discuss what *binary search* is, how it works, write its normal iterative code, and then use it recursively too.

Binary Search Technique

This popular search technique searches the given *ITEM* in minimum possible comparisons. The *binary search* requires **the array**, to be scanned, **must be sorted in any order** (for instance, say ascending order). In binary search, the *ITEM* is searched for in a smaller *segment* (nearly half the previous segment) after every stage. For the first stage, the segment contains the entire array.

To search for *ITEM* in a sorted array (in *ascending order*), the *ITEM* is compared with *middle element* of the segment (*i.e.*, in the entire array for the first time). If the *ITEM* is more than the *middle element*, latter part of the segment becomes new segment to be scanned ; if the *ITEM* is less than the *middle element*, former part of the segment becomes new segment to be scanned. The same process is repeated for the new segment(s) until either the *ITEM* is found (search successful) or the segment is reduced to the single element and still the ITEM is not found (search unsuccessful).

NOTE

Binary search can work for only sorted arrays whereas linear search can work for both sorted as well as unsorted arrays.

Following figure illustrates the process of binary search in a sorted array. Search item is Key in a sorted array with N elements.

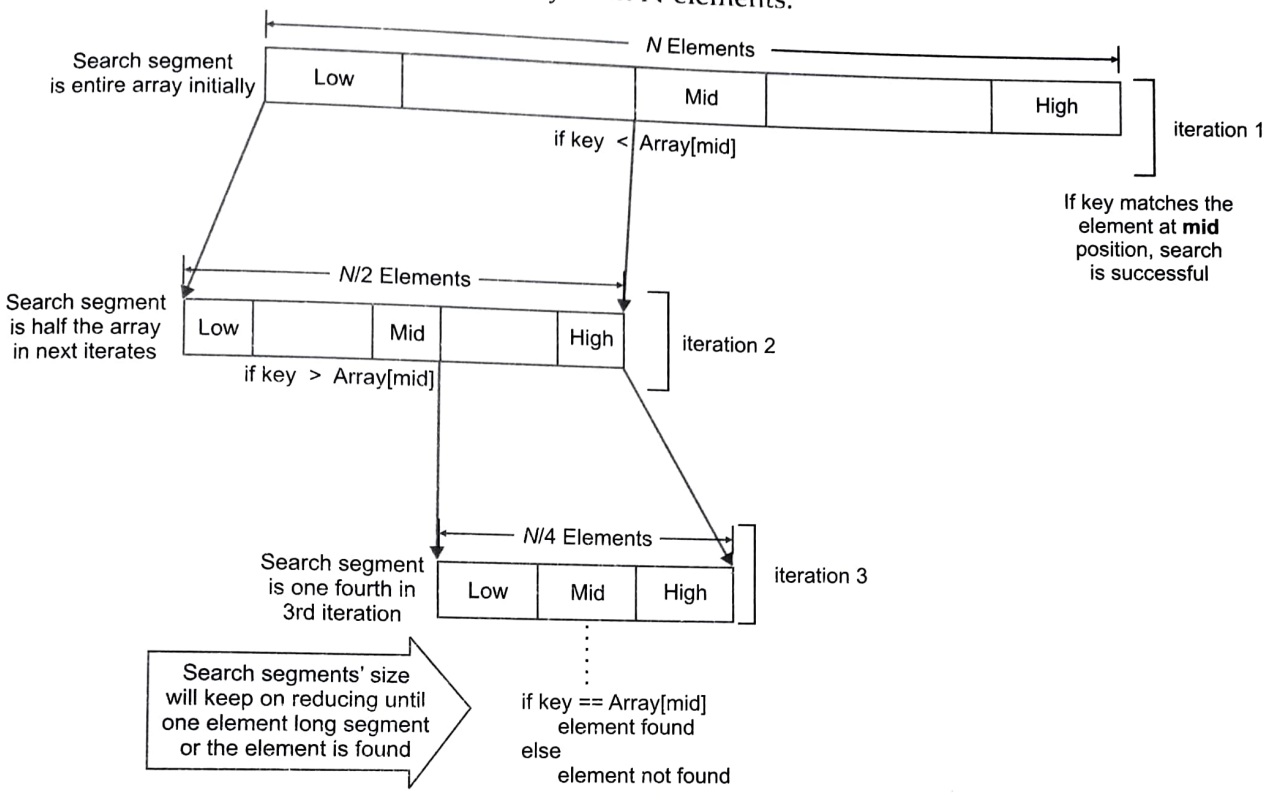


Figure 6.4 Working of Binary Search Algorithm.

Algorithm *Binary Search in Linear List (Array AR[L : U])*

Case I : AR in *ascending* order

```
# Initialise segment variables
1.   Set beg = L, last = U    # L is 0, U is size-1
2.   while beg <= last, perform steps 3 to 6
3.        mid = (beg + last)/2
4.        if AR [mid] == ITEM then
          {   print("Search Successful")
              print(ITEM, "found at", mid)
              break
          }
5.        if AR[mid] < ITEM then
              beg = mid + 1
6.        if AR[mid] > ITEM then
              last = mid – 1
     # End of while
7.   if beg ≠ last
         print("Unsuccessful Search")
8.   END.
```

Case II : AR in *descending order*

```
# Initialise
:
if AR[mid] = =ITEM then
{    :
}
if AR[mid] < ITEM then
     last = mid – 1
if AR[mid] > ITEM then
     beg = mid + 1
:
# Rest is similar to the algorithm in Case I.
```

NOTE

beg, last signify the limits of the search segment. Sometimes **beg, last** are also termed as **low** and **high** to signify lower and higher limit of the search segment.

Program 6.7 Binary Searching in an array (a sorted list).

```
def binsearch(ar, key) :
    low = 0                          # initially low end is at 0
    high = len(ar) - 1               # initially high end is at size -1
    while low <= high :
        mid = int((low + high) / 2)
        if key == ar[mid] :          #if key matches the middle element
            return mid               # then send its inde in array
        elif key < ar[mid] :
            high = mid - 1           # now the segment should be first half
        else:
            low = mid + 1            # now the segment should be latter half
    else:    # loop's else
        return -999

# __main__
ar = [12, 15, 21, 25, 28, 32, 33, 36, 43, 45]
item = int(input("Enter search item :"))
res = binsearch(ar, item)
if res >= 0 :        # if res holds a 0..n value
    print(item,"FOUND at index", res )
else:
    print("Sorry!", item, "NOT FOUND in array")
```

```
Enter search item: 32
32 FOUND at index 5
==============================
Enter search item: 15
15 FOUND at index 1
==============================
Enter search item: 10
Sorry! 10 NOT FOUND in array
==============================
Enter search item: 45
45 FOUND at index 9
==============================
Enter search item: 12
12 FOUND at index 0
==============================
Enter search item: 35
Sorry! 35 NOT FOUND in array
```

Some sample runs of the above program as shown on the right.

Pre-requisits of Binary Search

In order to implement binary search on an array, following conditions must be fulfilled.

(*i*) the given array or sequence must be sorted.

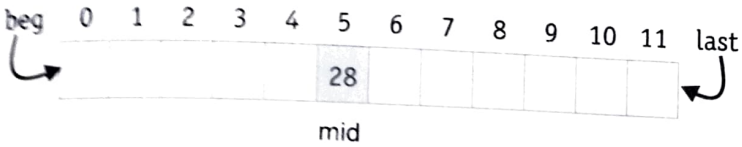
(*ii*) its sort-order and size must be known.

Binary search is very fast compared to linear search on one-dimensional arrays. However, linear search can work on both sorted and unsorted arrays while binary search can work with sorted arrays only. Also, binary search algorithm works with single-dimensional array where linear search algorithm can work with single and multi-dimensional arrays.

Example 6.1 *Write the steps to search 44 and 36 using binary search in the following array DATA :*

10	12	14	21	23	28	31	37	42	44	49	53
0	1	2	3	4	5	6	7	8	9	10	11

Solution. (*i*) **Search for 44.**



Step I :

beg = 0 ; last = 11

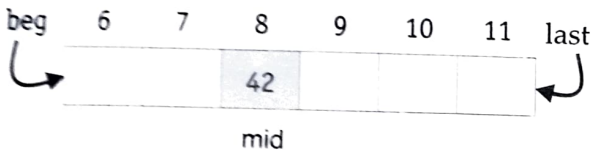
$$\text{mid} = \text{INT}\,\frac{(0+11)}{2} = \text{Int}\,(5.5) = 5$$

Step II :

Data[mid] *i.e.,* Data[5] is 28

28 < 44 then

beg = mid + 1 *i.e.,* beg = 5 + 1 = 6

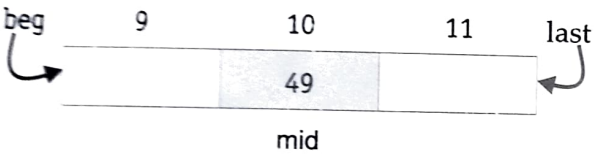


Step III :

$$\text{mid} = \text{INT}\,((\text{beg} + \text{last})/2) = \text{INT}\left(\frac{(6+11)}{2} = 8\right)$$

Data[8] *i.e.,* 42 < 44 then

beg = mid + 1 *i.e.,* beg = 8 + 1 = 9



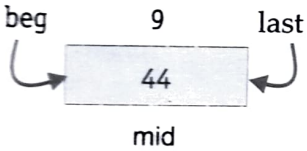
Step IV :

$$\text{mid} = \text{INT}\left(\frac{9+11}{2}\right) = 10$$

Data[10] *i.e.,* 49 > 44 then

last = mid - 1

last = 10 − 1 = 9



Step V :

$$\text{mid} = \text{INT}\left(\frac{(\text{beg} + \text{last})}{2}\right) = \frac{(9+9)}{2} = 9$$

(beg = last = 9)

Data [9] *i.e.,* 44 = 44

SEARCH SUCCESSFUL !! At location number 10.

(*ii*) **Search for 36**

Step I : beg = 0 ; last = 11

$$\text{mid} = \text{INT}\left(\frac{0+11}{2}\right) = 5$$

Step II : Data [mid] *i.e.,* Data [5] is 28

28 < 36 then beg

= mid + 1 = 5 + 1 = 6

Step III : $\text{mid} = \text{INT}\left(\frac{6+11}{2}\right) = 8$

Data[8] *i.e.,* 42

42 > 36 then

last = mid − 1 = 8 − 1 = 7

Step IV : $\text{mid} = \text{INT}\left(\frac{6+7}{2}\right) = 6$

Data [6] is 31

31 < 36 then

beg = mid + 1 = 6 + 1 = 7

Step V : $\text{mid} = \text{INT}\left(\frac{7+7}{2}\right) = 7$

(beg = last = 7)

Data [7] is 37 $\Rightarrow$ 37 ≠ 36

SARCH UNSUCCESSFUL !!

6.4.3 Recursive Binary Search

Now that you have fair idea about Binary search algorithm and how it works, let us see how it can be implemented recursively.

As you can see that process of finding the element is same in all search-segments, only the lower limit *low* and higher limit *high* of a segment changes if the element is not found at the middle (*mid*) position of the search-segment. Thus, a recursive call can be made to the binary search function by changing the limits. The search stops when either the element is found and its index is returned OR lower limit becomes more than higher limit–this will happen when the search- segment reduces to size of single element and search is still unsuccessful, in this case both mid + 1 or mid – 1 will make lower limit more than higher limit - which means search is unsuccessful.

Following program lists the recursive version of binary search algorithm.

Program 6.8 Write a recursive function to implement binary search algorithm.

```
# binary recursive search
def binsearch(ar, key, low, high) :
    if low > high :                    # search unsuccessful
        return -999                    BASE CASES
    mid = int((low + high) / 2)

    if key == ar[mid] :                # if key matches the middle element
        return mid                     # then send its index in array
    elif key <ar[mid] :
        high = mid -1                  # now the segment should be first half
        return binsearch(ar, key, low, high)   RECURSIVE CASES
    else:
        low = mid + 1                  # now the segment should be latter half
        return binsearch(ar, key, low, high)
```

```
# __main__
ary = [12, 15, 21, 25, 28, 32, 33, 36, 43, 45]
                                # sorted array
item = int(input("Enter search item:"))
res = binsearch(ary, item, 0, len(ary)-1)
if res >= 0 :   # if res holds a 0..n value,
    print(item,"FOUND at index", res)
else:
    print("Sorry!", item, "NOT FOUND in array")
```

Some sample runs of the above program as shown on the right .

```
Enter search item: 32
32 FOUND at index 5
==============================
Enter search item: 15
15 FOUND at index 1
==============================
Enter search item: 10
Sorry! 10 NOT FOUND in array
==============================
Enter search item: 45
45 FOUND at index 9
==============================
Enter search item: 12
12 FOUND at index 0
==============================
Enter search item: 35
Sorry! 35 NOT FOUND in array
```

RECURSION PRACTICALLY

riP Progress In Python 6.1

This 'PriP' session is aimed at practice of recursion in Python.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 6.1 under Chapter 6 after practically doing it on the computer.

>>>❖<<<

Check Point 6.1

Which of the following are examples of recursive functions ?

(*a*)
```
def Print(ch) :
    if ch != ' ' :
        print(ch + 1)
Print('k')
```

(*b*)
```
def recur(p) :
    if p == 0 :
        print("##")
    else:
        recur(p)
        p = p - 1
    recur(5)
```

(*c*)
```
def recur(p) :
    if p == 0 :
        print("#")
    else:
        p = p - 1
        recur1(p)
def recur1(n) :
    if n % 2 == 0:
        return n
    else:
        return (n-5)
```

(*d*)
```
def Check(c) :
    Mate(c+1)
def Mate(d) :
    Check(d-1)
```

(*e*)
```
def PrnNum(n) :
    if n == 1 :
        return
    else :
        print(n)
        PrnNum(n-2)
```

6.5 RECURSION VS ITERATION

Recursion and loops are actually related concepts. Generally, anything you can do with a loop, you can do with recursion, and vice versa. Sometimes one way is simpler to write, and sometimes the other is, but in principle they are interchangeable. Although loop and recursion are interchangeable, yet there are some examples where recursion is indeed the best way to approach a problem.

Even for problems that can be treated equally well through iteration and recursion, there is a subtle difference which is because of the way loops and method calls are treated by a programming language compiler.

When a loop repeats, it uses same memory locations for variables and repeats the same unit of code. Whereas in recursion, instead of repeating the same unit of code and using the same memory locations for variables, fresh memory space is allocated for each recursive call.

As it happens, any problem that can be solved via iteration can be solved using recursion and any problem that can be solved via recursion can be solved using iteration. Iteration is preferred by programmers for most recurring events, reserving recursion for instances where the programming solution would be greatly simplified. In a programming language, recursion involves an additional cost in terms of the space used in RAM by each recursive call to a function and in time used by the function call.

Because of extra memory stack manipulation, recursive versions of functions often run slower and use more memory than their iterative counterparts. But this is not always the case, and recursion can sometimes make code easier to understand.

NOTE

Recursion is heavy on memory because with each recursive call, there are unfinished function calls still lying in memory. Memory use decreases only after base case has been reached and unfinished function calls start getting completed.

So, you can say that :

- recursion makes a solution look shorter, closer in spirit to abstract mathematical entity.
- iteration makes a solution less costly to implement but all logic is part of loop, so lengthier it appears.

It always depends on the problem being solved which approach is better for it – iteration or recursion.

LET US REVISE

- *A function is said to be recursive if it calls itself.*
- *There are two cases in each recursive functions ; the recursive case and the base case.*
- *The base case is the case whose solution is pre-known and is used without computation.*
- *The recursive case is more general case of problem, which is being solved.*
- *An infinite recursion is when a recursive function calls itself endlessly.*
- *If there is no base case, or if the base case is never executed, infinite recursion occurs.*
- *Iteration uses same memory space for each pass contrary to recursion where fresh memory is allocated for each successive call.*
- *Recursive functions are relatively slower than their iterative counterparts.*
- *Some commonly used recursive algorithms are factorial, gcd, fibonacci series printing, binary search etc.*

Objective Type Questions — OTQs

Multiple Choice Questions

1. Recursion is a _____ .
   (*a*) generic class.
   (*b*) process of setting a value based on its previous value.
   (*c*) process of defining a method that calls itself.
   (*d*) process of repeatedly calling other methods.
2. A function that is called by itself, either directly or indirectly is called as
   (*a*) Super function
   (*b*) Recursive function
   (*c*) Main function
   (*d*) All of these
3. Iteration uses a repetition/looping structure whereas recursion uses
   (*a*) Sequence structure
   (*b*) Selection structure
   (*c*) Looping structure
   (*d*) Function call
4. Recursion is heavy on memory as compared to iteration because
   (*a*) There are multiple cases in recursive functions.

(b) There are unfinished function calls lying in memory.
(c) Loops consume lesser memory than multiple cases.
(d) Functions can run faster after consuming more memory.

5. In a recursive function, the case for which solution is pre-known or computable with an expression is called _____ .
   (a) Recursive case
   (b) Explicit case
   (c) Base case
   (d) repetitive case

6. In a recursive function, _____ case must always be reachable.
   (a) Recursive case
   (b) Explicit case
   (c) Base case
   (d) repetitive case

7. In a recursive function, _____ case always invokes another function.
   (a) Recursive case
   (b) Explicit case
   (c) Base case
   (d) repetitive case

8. Which of the following is correct skeleton for a recursive function ?

   (a)
```
def solution (N) :
    if base_case_condition :
        return something easily computed or directly available
    else :
        divide problem into pieces
        return something calculated using solution(some number)
```

   (b)
```
def solution (N) :
    if base_case_condition :
        return something easily computed or directly available
    else :
        divide problem into pieces
        return something calculated using solution(N)
```

   (c)
```
def solution (N) :
    divide problem into pieces
    return something calculated using solution(N)
```

   (d)
```
def solution (N) :
    if base_case_condition :
        return something easily computed or directly available
    else :
        divide problem into pieces
        return something calculated using solution(some number other than N)
```

9. In a recursive function, if the base case is not reached, what happens ?
   (a) Program executes and shows "Syntax error."
   (b) Program does not run at all.
   (c) Program executes the recursive case only once.
   (d) Program keeps on executing and "Out of Memory" or "Stack overflow" error occurs.

10. Which of the following cannot be converted in a recursive function ?
    (*a*) Factorial program
    (*b*) random function generation
    (*c*) Binary Search
    (*d*) Greatest Common Divisor (GCD)

11. Pick an option to complete the below given code which is computing factorial through a recursive function.

```
def factorial(n):
    if n == 0: # Base case
        return 1
    else:
        return _____________________ # Recursive call
```

    (*a*) $n*(n-1)$
    (*b*) n
    (*c*) $n*$factorial$(n-1)$
    (*d*) factorial$(n-1)*n$

12. What are the base cases in the following recursive function ?

```
def recfunction(n):
    if n > 0:
        print(n % 10)
        recfunction(n // 10)
```

    (*a*) n > 0
    (*b*) n <= 0
    (*c*) no base cases
    (*d*) n < 0

13. In the following function, what is the base case ?

```
def recfunction(n):
    if n == 1:
        return 1
    else
        return n + recfunction(n - 1)
```

    (*a*) n is 1
    (*b*) n is greater than 1
    (*c*) n is less than 1.
    (*d*) no base case.

14. Carefully read the following recursive function and choose one of the given options.

```
def factorial(n):
    return n * factorial(n - 1)
```

    (*a*) Invoking factorial(0) returns 0.
    (*b*) Invoking factorial(1) returns 0.
    (*c*) Invoking factorial(2) returns 2.
    (*d*) The function runs infinitely and causes a StackOverflowError.

15. How many times is the factorial function given below invoked for factorial(4) ?

```
def factorial(n):
    if n == 0: # Base case
        return 1
    else:
        return n * factorial(n - 1)
```

    (*a*) 3
    (*b*) 4
    (*c*) 5
    (*d*) 6

Fill in the Blanks

1. A function is said to be _____ if it calls itself.
2. The _____ Case in a recursive program must be reachable.
3. Every recursive function consists of one or more base cases and a general, _____ case.
4. When a recursive function calls itself endlessly, it is called _____ .
5. In a recursive function, the case with a determinable solution is called _____ _____ .
6. The recursive solution of a problem consumes _____ memory than the iterative solution of the problem.
7. Infinite recursion occurs when _____ case is not reached at all.

True/False Questions

1. There can be only one base case in a recursive code.
2. There can be multiple base cases in a recursive code but the base case must be reachable at some point of time.
3. Recursive solution is a compact solution than iterative solution of a problem.
4. Recursive solution is a faster solution than iterative solution of a problem.
5. Recursive solution consumes same memory as the iterative solution of a problem.
6. Every recursive call reduces the original problem into a simpler form and each successive calls brings it closer to a base case.
7. Every recursive function must have a return value.
8. A recursive function is invoked differently than a non-recursive function.
9. Recursive function calls another function before finishing previous function call.
10. Infinite recursion may eat up all the available stack memory.

NOTE : Answers for OTQs are given at the end of the book.

Solved Problems

1. *What is recursion ?*

Solution. In a program, if a function calls itself (whether directly or indirectly), it is known as recursion. And the function calling itself is called recursive function *e.g.*, following are two examples of recursion :

(*i*)
```
def A( ) :
    A( )
```

(*ii*)
```
def B( ) :
    C( )
def C( ) :
    B( )
```

2. *What are base case and recursive case? What is their role in a recursive program?*

Solution. In a recursive solution, the **Base cases** are predetermined solutions for the simplest versions of the problem : if the given problem is a base case, no further computation is necessary to get the result.

The **recursive case** is the one that calls the function again with a new set of values. The recursive step is a set of rules that eventually reduces all versions of the problem to one of the base cases when applied repeatedly.

3. *Which of the following is correct skeleton for a recursive function?*

(*a*)
```
def solution (N) :
    if base_case_condition :
        return something easily computed or directly available
    else :
        divide problem into pieces
        return something calculated using solution(some number)
```

(*b*)
```
def solution (N) :
    if base_case_condition :
        return something easily computed or directly available
    else :
        divide problem into pieces
        return something calculated using solution(N)
```

(*c*)
```
def solution (N) :
    divide problem into pieces
    return something calculated using solution(N)
```

(*d*)
```
def solution (N) :
    if base_case_condition :
        return something easily computed or directly available
    else :
        divide problem into pieces
        return something calculated using solution(some number other than N)
```

Solution. (*d*)

4. *Why is base case so important in a recursive function ?*

Solution. The base case, in a recursive case, represents a pre-known case whose solution is also preknown. This case is very important because upon reaching at base case, the termination of recursive function occurs as base case does not invoke the function again, rather it returns a pre-known result. In the absence of base case, the recursive function executes endlessly. Therefore, the execution of base case is necessary for the termination of the recursive function.

5. *When does infinite recursion occur ?*

Solution. Infinite recursion is when a recursive function executes itself again and again, endlessly. This happens when either the base case is missing or it is not reachable.

6. *Compare iteration and recursion.*

Solution. In iteration, the code is executed repeatedly using the same memory space. That is, the memory space allocated once, is used for each pass of the loop.

On the other hand in recursion, since it involves function call at each step, fresh memory is allocated for each recursive call. For this reason *i.e.*, because of function call overheads, the recursive function runs slower than its iterative counterpart.

7. *State one advantage and one disadvantage of using recursion over iteration.*

Solution. **Advantage.** Recursion makes the code short and simple while iteration makes the code longer comparatively.

Disadvantage. Recursion is slower than iteration due to overhead of multiple function calls and maintaining a stack for it.

8 ***What will the following function Check( ) return when the values of both 'm' and 'n' are equal to 5 ? Show the working.***

```
def Check(m, n) :
    if n == 1 :
        return -m
    else:
        return (m + 1) + Check(m + 1, n - 1)
```

Solution.

Start	m	n	Statement executed	Internal work and Stack	
Call 1	5	5	Check (m, n) *i.e.*, Check (5, 5) n == 1	5 == 1 False	
Call 2	6	4	return (m + 1) + Check(m, n – 1)	return 6 + **Check (6, 4)**	6 + **Check (6, 4)**
			Check (6, 4) n == 1	4 == 1 = False	
Call 3	7	3	return (m + 1) + Check(m, n – 1)	return 7 + **Check(7, 3)**	7 + **Check(7, 3)** 6 + **Check (6, 4)**
			Check(7, 3) n == 1	3 == 1 = False	
Call 4	8	2	return (m + 1) + Check (m, n – 1)	return 8 + **Check(8, 2)**	8 + **Check(8, 2)** 7 + **Check(7, 3)** 6 + **Check (6, 4)**
			Check(8, 2) n == 1	2 == 1 = False	
Call 5	9	1	return (m + 1) + Check(m, n – 1)	return 9 + **Check(9, 1)**	9 + **Check(9, 1)** 8 + **Check(8, 2)** 7 + **Check(7, 3)** 6 + **Check (6, 4)**
			Check(9, 1)		
Call 6	8		n == 1 return – m ;	1 == 1 = True **– 9** and returns to **call 5** statement *i.e.*, Check (9, 1)	–9 9 + (–9) = 0 8 + **Check(8, 2)** 7 + **Check(7, 3)** 6 + **Check (6, 4)**
Call 5			return 9 + (– 9)	returns 0 to **Call 4** *i.e.*, Check(8, 2)	0 8 + (0) = 8 7 + **Check(7, 3)** 6 + **Check (6, 4)**
Call 4			return 8 + 0	returns 8 to Call 3 *i.e.*, Check(7, 3)	8 7 + **(8) = 15** 6 + **Check (6, 4)**
Call 3			return 7 + 8	returns 15 to Call 2 *i.e.*, Check(6, 4)	15
Call 2			return 6 + 15	returns 21 to Call 1	6 + **(15) = 21**
Call 1			**21** = Ans		

9. *Consider the following function that takes two positive integer parameters x and y. Answer the following questions based on the code below.*

```
def compute (x, y) :
    if x > 1:
        if x % y == 0:
            print(y, end = ' ')
            compute (int(x/y), y)
        else:
            compute (x, y + 1)
```

(*a*) *What will be printed by the function call* **compute (24, 2)** *?*

(*b*) *What will be printed by the function call* **compute (84, 2)** *?*

(*c*) *State in one line what is* **compute( )** *trying to calculate ?*

Solution.

(*a*) 2 2 2 3

(*b*) 2 2 3 7

(*c*) Finding factors of **x** which are >= **y**

10. *Consider the following recursive function which has a base case defined and recursive case too. But when run with an odd number as parameter, this always gives* **RecursionError**. *Figure out the reason and suggest the solution.*

```
def skip_prod(n):
    """Return the product of n * (n - 2) * (n - 4) * ... """
    if n == 0 :                          # the base case
        return 1
    else:
        return n * skip_prod(n - 2)      # recursive case
```

Solution. In a recursive code, the base case should always be available and reachable too. That means, it must get executed for some value of the parameter passed.

Let us consider what happens when we choose an odd number for ***n***, *e.g.*,

skip_prod(3) will return **3 * skip_prod(1).**
skip_prod(1) will return **1 * skip_prod(-1).**

Here arises the problem. Since parameter ***n*** decreases by 2 at a time, for odd numbers, it will completely miss **the base case**, as from 1, the value of *n* will decrease to –1 completely missing ***n* == 0, the base case** ; and the function will end up recursing indefinitely.

The solution to this case is that the values less than 1 should also be considered. Thus the corrected code may be like : (there may be other solutions as well)

```
def skip_prod(n):
    if n <= 0 :
        return 1
    else:
        return n * skip_prod(n - 2)
```

Now, base case is reachable for all values

11. *Figure out the error in the following recursive code of factorial :*

```
def factorial(n) :
    if n == 0:
        return 1
    else:
        n * factorial(n-1)
#__main__
print(factorial(4) )
```

Solution. The error in above code is that the recursive case is calling function factorial( ) with a changed value but not returning the result of the call.

The corrected version will be :

```
def factorial(n) :
    if n == 0:
        return 1
    else:
        return n * factorial(-1)
#__main__
print(factorial(4))
```

12. *Why is following code printing 1 endlessly ?*

```
def Out_upto(n):
    i = 1
    if i > n:
        return
    else:
        print(i)
        i += 1
        Out_upto(i)
Out_upto(4)
```

Solution. The above code is printing 1 endlessly because the base case **i > n** is never reachable. The reason is that value of *i* becomes 1 for each call to function *Out_upto( )*.

13. *Write your own version of code so that the problem in previous question gets solved.*

Solution.

```
def Out_upto(n):
    if n == 0 :
        return
    else:
        Out_upto(n-1)
        print(n)
#__main__
Out_upto(4)
```

14. *Write recursive code to compute and print sum of squares of n numbers. Value of n is passed as parameter.*

Solution.

```
def sqsum(n):
    if n == 1:
        return 1
    return n * n + sqsum(n - 1)
#__main__
n = int(input("Enter value of n :"))
print(sqsum(n))
```

15. *Write recursive code to compute greatest common divisor of two numbers.*

Solution.

```
def gcd(a,b):
    if(b == 0):
        return a
    else:
        return gcd(b, a % b )
n1 = int(input("Enter first number:"))
n2 = int(input("Enter second number:"))
d = gcd(n1, n2)
print("GCD of", n1, "and", n2, "is:", d)
```

16. *Write a Recursive function in python* ***BinarySearch(Arr,l,R,X)*** *to search the given element* ***X*** *to be searched from the List* ***Arr*** *having* ***R*** *elements, where l represents lower bound and* ***R*** *represents the upper bound.*

[CBSE Sample Paper 2019-20]

Solution.

```
def BinarySearch (Arr, l, R, X):
if R >= l:
    mid = l + (R-l)//2
    if Arr[mid] = = X:
        return mid
    elif Arr[mid] > X:
        return BinarySearch(Arr, l, mid - 1, X)
    else:
        return BinarySearch(Arr, mid + 1, r, X)
    else:
        return -1
Arr = [ 2, 3, 4, 10, 40 ]
X = int(input('Enter element to be searched'))
result = BinarySearch(Arr, 0, len(Arr)-1, X)
if result != -1:
    print ("Element is present at index", result)
else:
    print ("Element is not present in array")
```

17. *Write a Recursive function* **recurfactorial(n)** *in python to calculate and return the factorial of number* ***n*** *passed to the parameter.*

Solution. [CBSE Sample Paper 2019-20]

```
def recurfactorial(n):
    if n == 1:
        return n
    else:
        return n*recurfactorial(n-1)
num = int(input("Enter a number:"))
if num < 0:
    print("Sorry, No factorial for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    print("The factorial of",num,"is",recurfactorial(num))
```

GLOSSARY

Recursion	Process of calling a function from within itself.
Recursive function	A function calling itself.
Infinite Recursion	A recursive function executing endlessly.
Base Case	Preknown case of recursive function, whose solution is pre-known.
Recursive Case.	The case of recursive function which invokes the function again.

Assignments

For Selective Assignment Solutions

Scan QR Code

Type A : Short Answer Questions/Conceptual Questions

1. What is a recursive function ? Write one advantage of recursive functions.
2. What is direct recursion and indirect recursion ?
3. What are the two cases required in a recursive function ?
4. What is base case ?
5. What is recursive case ?
6. Is it necessary to have a base case in a recursive function ? Why/Why not ?
7. What is infinite recursion ? Why does it occur ?
8. How can you stop/resolve an infinite recursion ?
9. Give some examples that can be represented recursively.
10. Can each recursive function be represented through iteration ? Give examples.
11. Which of the functions given in previous question will result into infinite recursion ?

12. Identify the base case(s) in the following recursive function :

```
def function1(n) :
    if n == 0 :
        return 5
    elif n == 1 :
        return 8
    elif n > 0 :
        return function1(n-1) + function1(n-2)
    else:
        return -1
```

13. Why are recursive functions considered slower than their iterative counterparts ?
14. If any recursive function can be easily written using iterative code, then what is the need for recursion ? When would you prefer recursion over iteration and vice versa ?
15. Compare and contrast the use of iteration and recursion in terms of memory space and speed.

Type B : Application Based Questions

1. *Consider square numbers defined as follows :*

```
compute(1) = 1
compute(N) = compute(N-1) + 2N - 1
```

According to this definition, what is compute(3) ?

(*a*) compute(3) = compute(2) + compute(1)

(*b*) compute(3) = compute(2) - 2*3 +1

(*c*) compute(3) = compute(2) + 2*3 -1

(*d*) compute(3) = compute(3) + 2*3 -1

2. *Look at compute numbers again :*

```
compute(1) = 1
compute(N) = compute(N-1) + 2N -1
```

Which Python function below successfully implements this definition ?

(*a*)
```
def compute(N) :
    if N < 1 :
        return 1
    else:
        return N * N
```

(*b*)
```
def compute(N) :
    if N == 1 :
        return 1
    else:
        return compute(N - 1) + 2 * N - 1
```

(*c*)
```
def compute(N) :
    if N = 1 :
        return 1
    else:
        return compute(N - 1) + 2 * N – 1
```

(*d*)
```
def compute(N) :
    if N = 1 :
        return 1
    else:
        return compute( N )
```

3. *Look at compute numbers one more time :*

```
compute(1) = 1
compute(N) = compute(N-1) + 2N – 1
```

Assume the definition has been implemented correctly. How many invocations will there be on the call stack if __main__ calls compute(5) ?

(A) 1 (B) 3 (C) 5 (D) 6

4. Predict the output of following codes.

(a)
```
def codo(n):
    if n == 0:
        print('Finally')
    else:
        print(n)
        codo(n - 3)
codo(15)
```

(b)
```
def codo(n):
    if n == 0:
        print('Finally')
    else:
        print(n)
        codo(n - 2)
codo(15)
```

(c)
```
def codo(n):
    if n == 0:
        print('Finally')
    else:
        print(n)
        codo(n - 2)
codo(10)
```

(d)
```
def codo(n):
    if n == 0:
        print('Finally')
    else:
        print(n)
        codo(n - 3)
codo(10)
```

5. Predict the output of following code.

```
def express(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return express(x*x, n/2))
    else:
        return x * express(x, n-1)
express(2,5)
```

6. Consider following Python function that uses recursion :

```
def check(n) :
    if n <= 1 :
        return True
    elif n % 2 == 0 :
        return check(n/2)
    else :
        return check(n/1)
```

What is the value returned by check(8) ?

7. Can you find an error or problem with the above code? Think about what happens if we evaluate **check(3)**. What output would be produced by Python? Explain the reason(s) behind the output.

8. Consider the following Python function **Fn( )**, that takes an integer *n* parameter and works recursively as follows :

```
def Fn(n):
    print(n, end=" ")
    if n < 3:
        return n
    else:
        return Fn(n // 2) - Fn(n // 3)
```

What will be the result produced by following function calls ?

(*a*) Fn(12) (*b*) Fn(10) (*c*) Fn(7)

9. Figure out the problem with following code that may occur when it is run ?

```
def recur(p) :
    if p == 0 :
        print("##")
    else:
        recur(p)
        p = p-1
recur(5)
```

10. Check Point 6.1 has some recursive functions that cause infinite recursion. Make changes in the definitions of those recursive functions so that they recur finitely and produce a proper output.

Type C : Programming Practice/Knowledge based Questions

1. Write a function that takes a number and tests if it is a prime number using recursion technique.
2. Implement a function ***product( )*** to multiply 2 numbers recursively using + and – operators only.
3. The *hailstone sequence* starting at a positive integer *n* is generated by following two simple rules. If *n* is even, the next number in the sequence is ***n* / 2**. If ***n*** is odd, the next number in the sequence is **3*n + 1**. Repeating this process, the *hailstone sequence* gets generated. Write a recursive function **hailstone(n)** which prints the hailstone sequence beginning at *n*. Stop when the sequence reaches the number 1 (since otherwise, we would loop forever 1, 4, 2, 1, 4, 2, ...)
4. A happy number is a number in which the eventual sum of the square of the digits of the number is equal to 1.

Example : $28 = (2)^2 + (8)^2 = 4 + 64 = 68$

$68 = (6)^2 + (8)^2 = 36 + 64 = 100$

$100 = (1)^2 + (0)^2 + (0)^2 = 1 + 0 + 0 = 1$

Hence, 28 is a happy number.

Example : $12 = (1)^2 + (2)^2 = 1 + 4 = 5$

Hence, 12 is not a happy number.

Write a program that takes a number and checks if it is a happy number by using following two functions in it :

sum_sq_digits	:	returns the sum of the square of the digits of the number *x*, using the **recursive technique**
ishappy( )	:	checks if the given number is a happy number by calling the function **syn_sq_digits** and displays an appropriate message

5. A list namely **Adm** stores admission numbers of 100 students in it, sorted in ascending order of admission numbers. Write a program that takes an admission number and looks for it in list **Adm** using binary search technique. The binary search function should use recursion in it.