Chapter 6 Recursion

Ex 6.1 Solutions

1. Which of the following are examples of recursive functions ?

(a)
```
def Print (ch) :
        if ch != " " :
                print (ch+1l )
Print( 'k' )
```
Not a recursive function because function ' Print' is not calling itself
(b)
```
def recur (p ) ;
        if p == 0 :
                print ( "##" )
        else :
                recur( p)
                p = p-1
recur( S)
```
This is a recursive function as ' recur ' is calling itself, although this is infinite recursion
(c)
```
def recur (p) :
if p == 8 :
print ( "#" J
else :
p = p -1
recurl(p)
def recurl (n);
if n%2 == 0 :
return n
else :
return (n- 5 )
```
Not a recursive function because neither of the fun ct i ons is calling itself
(d)
```
def Check (e):
Mate(c+ 1 )
def Mate (d) :
Check(d- 1)
```

This is mutual recursion because the two functions are calling each other
```
def PrnNum(n):
if n== 1 :
return
else :
print (n)
PrnNum(n - 2 )
```
Recursive function

Back Exercise Part A

1. What is a recursive function? Write one advantage of recursive functions.

A recursive function is a function that calls itself. e.g.

```
def func (n):
if n== 0:
        return
func(n- 1 )
```

Recursion helps reduce the size of the program. It also avoids unnecessary calling of functions as only one function can do the whole task by repetitively calling itselt.

2. What is direct recursion and indirect recursion?

Direct recursion is a function calling itself directly from its function body. e.g.

```
def funct ():
        func()
```

Indirect recursion is a function calling another function which calls its caller function. e .g .

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

3. What are the two cases required in a recursion function?

The two cases required in a recursive function are;
Base case: result is known or computed without any recursive calling
Recursive case: the function calls itself recursively

4. What is base case?

Any condition where a recursive function does not invoke itself is called a base case. e.g.

```
def func ();
if n== 0:       # base case
        return 10
func(n- 1 )
```

5. What is recursive case?

The recursive case is where the function calls itself recursively unless the base case is met. e.g.

```
def func ();
        if n== 0:       # base case
                return 10
        func(n- 1 )     # recursive case
```

6. Is it necessary to have a base case in a recursive function? Why/Why not?

Yes, a base case is very important in a recursive function. Without it the code is not sensible, and the recursion will be infinite unless the maximum recursion limit is reached. e.g.

```
def func (); # There is no base case and the function will terminate with an error
        print ( ' Infinite recursion' )
        func ()
```

7. What is infinite recursion? Why does it occur?

Infinite recursion is when a function repetitively calls itself without terminating. It occurs when there is no base case, or if the base case is never executed. The function will eventually terminate with a maximum recursion limit exceeded exception. e.g.

```
def func ():
        print (‘Infinite recursion’)
        func()
```

8. How can you stop/resolve an infinite recursion?

To stop infinite recursion from occurring. make sure the function has a base case and the base case is met at some point, For instance

```
# if n is positive. base case will never get executed in the below function
def func (n) :
        if n== 0 :
                return 10
        func(n+1)
# base case will be met
def func (n):
        if n==0 :
                return 10
        func(n- l )
```

9. Give some examples that can be represented recursively.

Some problems which can be solved using recursion are:

i) sum of 1 to n

```
# Code
def func( n, s=0) :
        if n == 0 : # return sum when n is 0
                return s
        s += n # adds the current number
        return func(n- 1 , s)   # calls the function with n-1 and sum till now
print (func( 10 )) # 55
```

ii) factorial

```
def fac t (n):
        if n<= 1:
                return 1
        return n*fact(n- 1)
print (fact (6 )) # 720
```

10. Can each recursive function be represented through iteration? Give examples.

Generally, any program that can be written using recursion can also be written using iteration. e.g.
1) sum of 1 to n

```
# recursive
def func (n, s=0):
        if n== 0 : # return sum when n is 0
                return s
        S += n # adds the current number
        return func(n- 1 , s) #calls the function with n-1 and sum till now

# iterative
def func (n):
        s = 0
        for i in range (1, n+1):
                s += i
        return s
```

ii) factorial

```
# recursive
def fact (n):
        if n<= 1:
                return 1
        return n*fact(n- 1 )
# iterative
def fact (n):
        f = 1
        for i in range (1, n+1):
                f = f*i
        return f
```

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

Any case where the result is computed without a recursive call is called a base case. Base cases in the following function are:

```
def function1 (n):
        if n == 0 : # base case
                return 5
        elif n == 1 : # base case
                return 8
        elif n> 0:

                return functionl(n- 1) + functionl(n- 2 )

        else : # base case
                return - 1
```

13. Why are recursive functions considered slower than their iterative counterparts?

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. Recursion is slower because of extra memory stack manipulation. They are used because sometimes their use greatly simplifies the problem.

15. Compare and contrast the use of iteration and recursion in terms of memory space and speed.

Recursion involves an additional cost in terms of both the space used in RAM by each recursive call to a function and in time used by the function call. It is slower because of extra memory stack manipulation. They are used only because sometimes their use greatly simplifies the problem.

Back Exercise Part B

1. Compute 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
Ans :
(c) compute(3) = compute(2) + 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)

There will be 6 invocations, the first one being compute(5) and then each one being compute(N) = compute(N- l)+2*N- l for N from 5 to 1

4. Predict the output of following codes.

(a)
```
def codo (n) :
        if n == 0 :
                print ( ' Finally ' )
        else :
                print (n)
                codo(n - 3)
codo( 15)
# Output
15
12
9
6
3
Finally
```
(b)
```
def codo (n):
        if n == 0 :
                print ( 'Finally' )
        else :
                print (n)
                codo(n - 2)
codo(15)
# Output
RecursionError: maximum recursion depth exceeded while calling a Python object
# because n never becomes 0 and goes on to become negative
```
(c)
```
def codo (n):
        if n== 0:
                print ( ' Finally ' )
        else :
                print (n)
                codo(n - 2 )
codo( 10 )
#Output
8
6
4
```

```
2
Finally
( d )
def codo (n):
        if n == 0 :
                print ( ' Finally' )
        else :
                print (n)
                codo(n - 3 )
codo( 10 )
# Output
RecursionError : maximum recursion depth exceeded while calling a
Python object
# because n never becomes 0 and goes on to become negative
```

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)
print (express( 2 , 5 ))
#32
# The above code is trying to get pow( 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 )
print (check( 8 ) )
# True
# The above function wilt only work when the number is a power of 2.
# It win result in infinite recursion when n is not a power of 2.
```

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.

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

```
        return check(n/2)
    else :
        return check(n/1)
```

Yes, The above function will only work when the number is a power of 2. It will result in infinite recursion when n is not a power of 2. check(3) will result in an infinite recursion. The reason is that in the last else statement the function is calling itself without any change to the number n : check(n/1)

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 )
( a )
x = Fn( 12 )
# 12 6 3 1 1 2 4 2 1, x will be -3
( b )
y = Fn( 16 )
# 12 6 3 1 1 2 4 2 1 10 5 2 1 3 1 1 1, Y will be 1
( c )
z = Fn( 7)
# 12 6 3 1 1 2 4 2 1 10 5 2 1 3 1 1 7 3 1 1 2 -2, z will be -2
```

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)
# The above code will run into infinite recursion because the value
of p is being decremented after the recursion call. So it is never
actually changing. The correct way:
def recur (p) :
        if p== 0:
                print ( "##" )
        else :
                p = p-1
                recur(p)
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.

Checkpoint 6.1
(b)

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

The above code will run into infinite recursion because the value of p is being decremented after the recursion call. So it is never actually changing. The correct way:

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

(d)

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

The above code has no base case and will run into infinite recursion. It can be corrected as following

```
def Check(c):
   if c > 10: # Any base case can be added
      return
   else:
      c = c+1 # Increment is important
Mate(c+1)
def Mate(d) :
Check(d-1)
```

Back Exercise Part C

1. Write a function that takes a number and tests if it is a prime number using recursion technique.

```
def prime (n , i = 2 ) :
# Base cases
if (n <= 2 ):
return True if (n == 2) else False
if (n % i == 0 ):
return False
```

```
if (i*i >n):
return True
# Check for next divisor
return prime(n, i + 1)
print (prime(15 ))
print (prime( 23 ) )
# Output
False
True
```

2. Implement a function product( ) to multiply 2 numbers recursively using + and – operators only.

```
# function to multiply two positive integers
def prod (a, b, p=0 );
        p += a # add a to the product b number of times
        if b ==1 :
                return p
        return prod(a, b- 1 , p)
print ( prod( 7 , 5))
print ( prod( 8 , 9 ))
# Output
35
72
```

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, ...)

```
def hail(n):
print(n, end=" ")
if n == 1:
return
if n%2 == 0:
hail(n//2)
else:
hail(3*n+1)

hail(7)
print()
hail(8)

# Output
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
8 4 2 1
```

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.

```
def sum sq digits (n): # function taking the sum of sq. of digits
        ans = 0
        for i in str (n):
                ans += int (i)* int (i)
        return ans
def is_happy (n):
        if ( len (str (n)) == 1:
                if n== 1:
                        return "Happy number"
                else :
                        return "Not a happy number"
        else :
                n = sum_sq_digits(n)    # n becomes sum of sq . of its digits
                return Is_happy(n)       # recursion
print (is_happy( 12 ))
print (is_happy( 28 ))
# Output
Not a happy number
Happy number
```

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.

```
def adm_search(arr, x, f=0, l=10):
if l >= f:
mid = (f+l)//2
if arr[mid] == x: # Found
return mid
elif arr[mid] > x:
return adm_search(arr, x, f, mid-1) # search below
else:
return adm_search(arr, x, mid+1, l) # search above
else:
return 'Not Found'

# adm list can be of any number
adm = [456, 467, 489, 500, 546, 567, 588, 599, 600, 612, 613]

print(adm_search(adm, 234))
print(adm_search(adm, 500))

# Output
Not Found
3
```