CHAPTER9

RECURSION

- Recursion.PropertiesofRecursion.
- Recursivefunctions:Factorials,RecursiveCallstack,TheFibonacciSequence.
- HowRecursionWorks-TheRunTimeStack.
- RecursiveApplications-RecursiveBinarySearch,TowersofHanoi.

NOTE:conceptof**recursivecallstack**explainedindetailinwhileexplainingruntimestack concept.

Afunction(ormethod)cancallanyotherfunction(ormethod),includingitself.Afunctionthatcallsitselfisknownasa**recursivefunction**.

Thefollowingimageshowstheworkingofarecursivefunctioncalledrecurse.

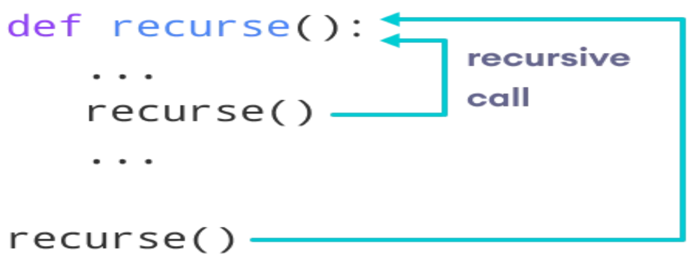


Fig:RecursiveFunctioninPython

Thefollowingtableoutlinesthekeydifferencesbetweenrecursionanditeration:

Recursion	Iteration
Terminates when a base case is reached	Terminates when a defined condition is met
Each recursive call requires space in memory	Each iteration is not stored in memory
An infinite recursion results in a stack overflow error	An infinite iteration will run while the hardware is powered
Some problems are naturally better suited to recursive solutions	Iterative solutions may not always be obvious

PropertiesofRecursion.

Allrecursivesolutionsmustsatisfythreerulesorproperties:

1. A recursivesolutionmustcontaina**basecase.**
2. Arecursivesolutionmustcontaina**recursivecase.**
3. Arecursivesolutionmustmake**progresstowardthebasecase.**

A recursive solution subdivides a problem into smaller versions of itself. For a problem to besubdivided,ittypicallymustconsistofadatasetoratermthatcanbedividedintosmallersets or a smaller term. This subdivision is handled by the recursive case when the function callsitself.

Thebase case istheterminating case andrepresentsthe smallest subdivision of the problem.Itsignals theendofthevirtualooporrecursivecalls.

Finally, a recursive solution must make progress toward the base case or the recursion willnever stop resulting in an infinite virtual loop. This progression typically occurs in eachrecursivecallwhenthelargerproblemisdividedintosmallerparts.

Recursivefunctions:

Factorials

Thefactorialofapositiveintegerncanbeusedtocalculatethenumberofpermutationsofnelements.Thefunctionisdefinedas:

$$n! = n * (n-1) * (n-2) * ... * 1$$

withthespecialcaseof0!=1.

Considerthefactorialfunctionondifferentintegervalues:

$0! = 1$
$1! = 2 * 1$
$2! = 3 * 2 * 1$
$3! = 4 * 3 * 2 * 1$
$4! = 5 * 4 * 3 * 2 * 1$

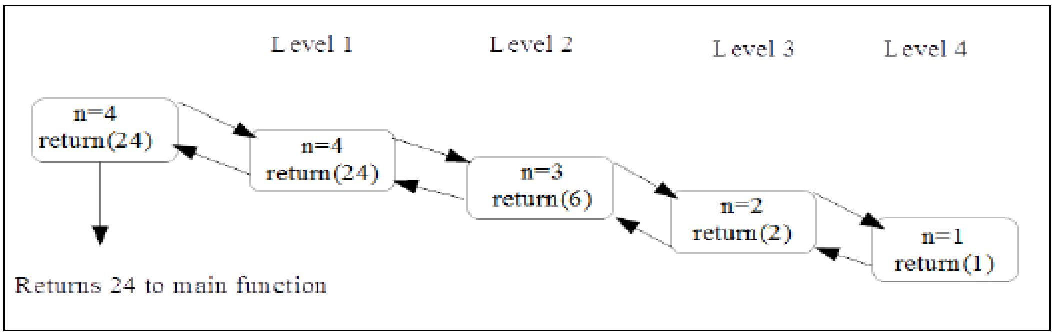
Aftercarefulinspectionoftheseequations,itbecomesobviouseachofthesuccessiveequations,forn >1,canberewrittenintermsofthepreviousequation:

0!=1
1!=1∗(1−1)!
2!=2∗(2−1)!
3!=3∗(3−1)!
4!=4∗(4−1)!
5!=5∗(5−1)!

Sincethefunctionisdefinedintermsofitselfandcontainsabasecase,arecursivedefinitionca nbeproducedforthefactorialfunctionasshownhere.

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

wecanvisualizetheprocessoffindingfactorialof4asshownbelow:



Program

Todemonstraterecursiveoperations(factorial)

```
deffactorial(n):i
    fn==0:
        return1
    return(n*factorial(n-1))

num=int(input("Enteranumbertofinditsfactorialvalue"))print("Factorialof",n
um,"is:",factorial(num))
```

OUTPUT

```
Enter a number to find its factorial value 5
Factorial of 5 is:  120
```

TheFibonacciSequence

TheFibonaccisequenceisasequenceofintegervaluesinwhichthefirsttwovaluesareboth1 and each subsequent value is the sum of the two previous values. The first 11 terms of thesequenceare:

1,1,2,3,5,8,13,21,34,55,89,...

ThenthFibonaccinumbercanbecomputedbytherecurrencerelation(forn>0):

```
            1                 if n = 0 or n = 1
F(n) = {
            F(n-1) + F(n-2)   otherwise
```

Program

Todemonstraterecursiveoperations(Fibonaccisequence)

```
deffib(n):
   if(n==0):r
      eturn0
   ifn==1orn==2:retu
      rn1
   else:
      returnfib(n-1)+fib(n-2)


n=int(input("Enteranumber"))
print("Fibonacciseriesof%dnumbersare:"%n,end="")foriinran
ge(0,n):
   print(fib(i),end="")
```

OUTPUT

```
Enter a number5
Fibonacci series of 5 numbers are : 0 1 1 2 3
```

HowRecursionWorks

Whenafunctioniscalled,thesequentialflowofexecutionisinterruptedandexecutionjumpstothebodyoft hatfunction.Whenthefunctionterminates,executionreturnstothepointwhereitwasinterruptedbefore thefunctionwasinvoked.

TheRunTimeStack

Each time a function is called, an activation record is automatically created in order to maintaininformation related to the function. One piece of information is the return address. This is thelocation of the next instruction to be executed when the function terminates. Thus, when afunction returns, the address is obtained from the activation record and the flow execution canreturntowhereitleftoffbeforethefunctionwascalled.

The activation records also include storage space for the allocation of local variables.Remember, a variable created within a function is local to that function and is said to have localscope. Localvariables arecreated when afunction begins execution andaredestroyedwhenthe function terminates. The lifetime of a local variable is the duration of the function in which itwascreated.

An activation record is created per function call, not on a per function basis. When a function iscalled,anactivationrecordiscreatedforthatcallandwhenitterminatestheactivationrecordisdestroy ed.

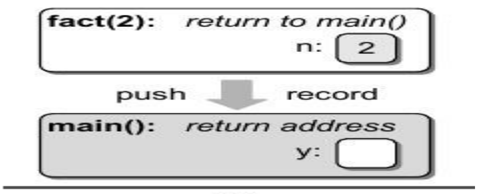
The system must manage the collection of activation records and remember the order in whichtheywerecreated.Itdoesthisbystoringtheactivationrecordsona**runtimestack**.

The run time stack is just like the stack structure but it’s hidden from the programmer and isautomaticallymaintained.

Consider the execution of the following code segment, which uses the factorial functiondefinedearlier:

Whenthemainroutineisexecuted,thefirstactivationrecordiscreatedandpushedontotherun time stack, as illustrated in Figure (a). When the factorial function is called, the secondactivationrecordiscreatedandpushedontothestack,asillustratedinFigure(b),andtheflowofex ecutionischangedtothatfunction.

(a)



(b)

Thefactorialfunctionisrecursivelycalleduntilthebasecaseisreachedwithavalueofn=0.Atthispoint,the runtimestackcontainsfouractivationrecords,asillustratedFigure(a).

When the base case statementis executed, the activation record for the function call fact(0) ispopped from the stack, as illustrated in Figure (b), and execution returns to the functioninstance fact(1). This process continues until all of the activation records have been poppedfromthestackandtheprogramterminates.

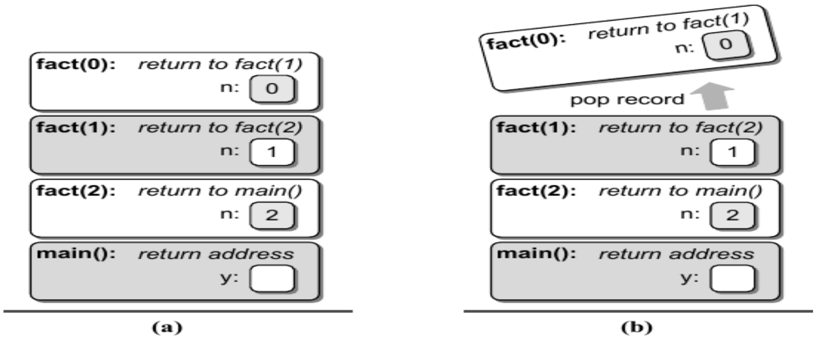


Figure:Theruntimestackforthesampleprogramwhenthebasecaseisreached.

RecursiveApplicationsR

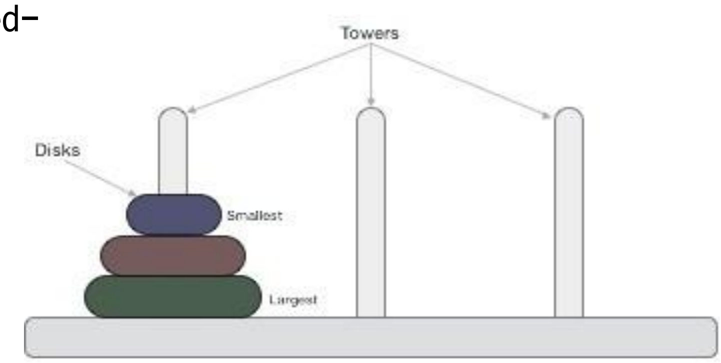
ecursiveBinarySearch

In searching for a target within the sorted sequence, the middle value is examined todetermineifitisthetarget.Ifnot,the sequence issplitinhalfandeitherthelowerhalfortheupper half of the sequence is examined depending on the logical ordering of the target value inrelation to the middle item. In examining the smaller sequence, the same process is repeated until either the target value is found or there are no additional values to be examined. Arecursiveimplementationofthebinarysearchalgorithmisprovidedbelow.

```
defrecBinarySearch(target,theSeq,first,last):iffir
  st>last:
    returnFalse
  else:
    mid=(last+first)//2
    iftheSeq[mid]==target:re
      turnTrue
    eliftarget<theSeq[mid]:
      returnrecBinarySearch(target,theSeq,first,mid-1)else:
      returnrecBinarySearch(target,theSeq,mid+1,last)
```

TowerofHanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more thanoneringsisasdepicted−



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller onesits over the larger one. There are other variations of the puzzle where the number of disksincrease,butthetowercountremainsthesame.

Rules

Themissionistomoveallthe disksstosomeanothertowerwithoutviolatingthesequenceofarrangement.Afewrulestobefollowedf orTowerofHanoiare−

- Onlyonediskcanbemovedamongthetowersatanygiventime.
- Onlythe"top"diskcanberemoved.
- Nolargediskcansitoverasmalldisk.

We divide the stack of disks in two parts. The largest disk (n^{th}disk) is in one part and all other(n-1)disksareinthesecondpart.Ourultimateaimistomoven thdiskfromsourcetodestinationandthenputallother(n-1)disksontoit.

Thestepstofolloware−

- Step1−Moven-1disksfromsourcetoaux
- Step2−Moventhdiskfromsourcetodest
- Step3−Moven-1disksfromauxtodest

ALGORITHM

START
ProcedureHanoi(disk,source,dest,aux)

```
  IFdisk==1,THEN
    movediskfromsourcetodestE
  LSE
    Hanoi(disk - 1, source, aux, dest) // Step
    1move disk from source to dest // Step
    2Hanoi(disk-1,aux,dest,source)//Step3
  ENDIF
```

ENDProcedure

PROGRAM

ImplementsolutionforTowersofHanoi.

```
defTowerOfHanoi(n,source,destination,auxiliary):
    ifn==1:
        print("Movedisk1fromsource",source,"todestination",destination)
        return
    TowerOfHanoi(n-1,source,auxiliary,destination)
    print
    ("Movedisk",n,"fromsource",source,"todestination",destination)TowerOf
    Hanoi(n-1,auxiliary,destination,source)

n =
4TowerOfHanoi(n,'A','B','C')
```

Output

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```