VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

RECURSION

Divide and Conquer

Recursion

- It is one of the most powerful tool in programming language. It is a process where function calls itself again and again.
- Recursion basically divides the big problem into small problems up to the point where it can be solved easily, for example if we have to calculate factorial of a 5, we will divide factorial of 5 as 5*factorial(4), then 4*factorial(3), then 3*factorial(2), then 2*factorial(1) and now factorial of 1 can be easily solved without any calculation, now each pending function will be executed in reverse order.

Condition for Implementing Recursion

- It must contain **BASE CONDITION** i.e. at which point recursion will end otherwise it will become infinite.
- **BASE CONDITION** is specified using 'if' to specify the termination condition
- Execution in Recursion is in reverse order using STACK. It first divide the large problem into smaller units and then starts solving from bottom to top.
- It takes more memory as compare to LOOP statement because with every recursion call memory space is allocated for local variables.
- The computer may run out of memory if recursion becomes infinite or termination condition not specified.
- It is less efficient in terms of speed and execution time
- Suitable for complex data structure problems like TREE, GRAPH etc

Recursion – To print message forever

```
def message():
        print("Hi There!")
        message()

message()
```

In this code, BASE CONDITION is missing so message() function will call it self again and again till out of memory error

Example 1 – Recursion (Factorial)

```
def factorial(num):
    if num == 1:
        return 1
    else:
        return num * factorial(num-1)

n = int(input("Enter any number "))
f = factorial(n)
print("Factorial of ", n, " is ", f)
```

```
Enter any number 5
Factorial of  5  is  120
```

First time num=5, so goes to else:
return 5 * factorial(4) # pushed to Stack..(i)
Second time num=4, so goes to else:
return 4 * factorial(3) # pushed to Stack..(ii)
Third time num=3, so goes to else:
return 3 * factorial(2) # pushed to Stack..(iii)
Fourth time num=2, so goes to else:
return 2 * factorial(1) # pushed to Stack..(iv)
Fifth time num=1, so goes to if:
return 1 # goes to (iv)
then goes to (iii) then goes to (ii) then goes to (i) and finally back to calling function

Example -2 (Sum of number between 1 to n)

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

n = int(input("Enter upper range to calculate sum "))
t = sum_n(n)
print("Sum of 1 to ",n, " is ", t)
```

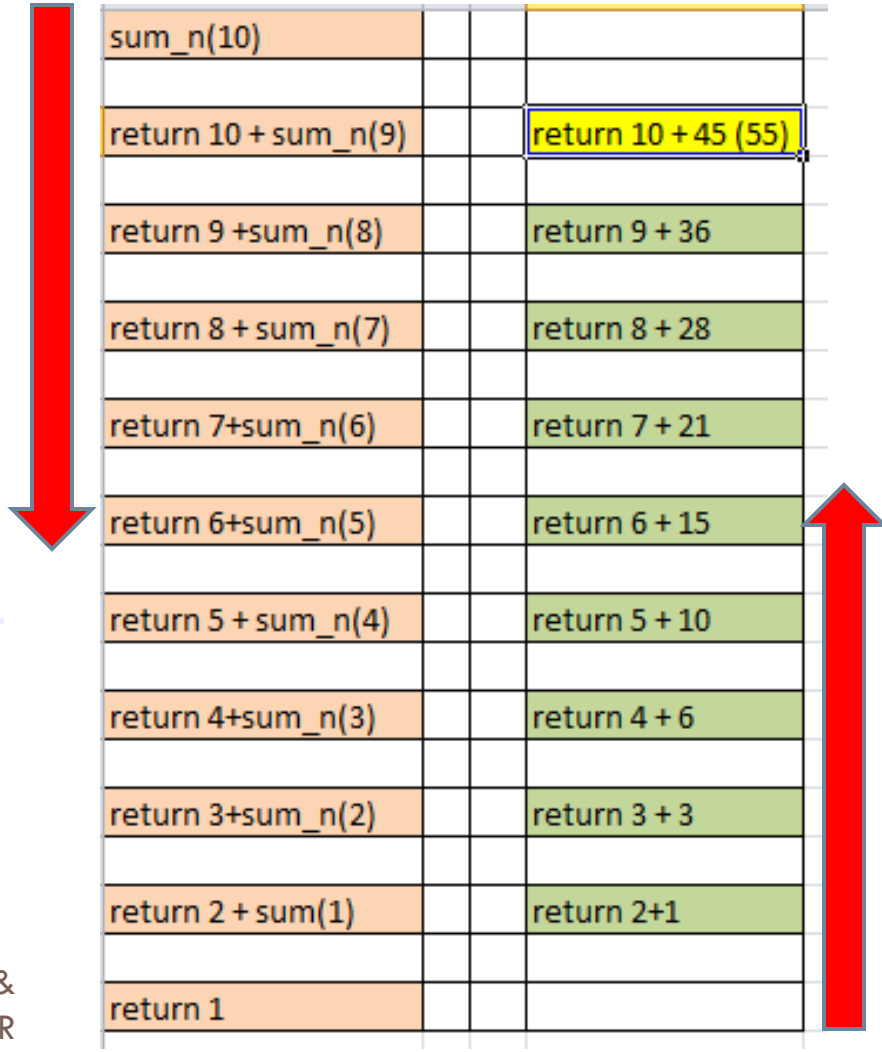
```
Enter upper range to calculate sum 10
Sum of 1 to  10  is  55
```

Example -2 (Sum of number between 1 to n)

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

n = int(input("Enter upper range to calculate sum "))
t = sum_n(n)
print("Sum of 1 to ",n, " is ", t)
```

```
Enter upper range to calculate sum 10
Sum of 1 to  10  is  55
```



VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

Example -3 (Sum of number between 1 to n) using Iteration

```
def sum_n(n):
    t = 0
    for i in range(1, n+1):
        t += i
    return t

n = int(input("Enter upper range to calculate sum "))
t = sum_n(n)
print("Sum of 1 to ",n, " is ", t)
```

```
Enter upper range to calculate sum 10
Sum of 1 to  10  is  55
```

Example 4 - Recursion

```
def fibonacci(num):
    if num <= 1:
        return num
    else:
        return (fibonacci(num-1)+fibonacci(num-2))

n = int(input("Enter how many terms "))
for i in range(n):
    print(fibonacci(i))
```

```
Enter how many terms 10
0
1
1
2
3
5
8
13
21
34
```

How Fibonacci series is executing?

Suppose the value of n is entered as 10, then					
					OUTPUT
i=0	fibonacci(0)	return 0	because num<=1		0
i=1	fibonacci(1)	return 1	because num<=1		0 1
i=2	fibonacci(2)	return fibonacci(1) + fibonacci(0)			
		i.e. return 1 + 0 which is 1			0 1 1
i=3	fibonacci(3)	return fibonacci(2) + fibonacci(1)			
	taking first call	return fibonacci(1) + fibonacci(0)			
		return 1			
	taking second call	fibonacci(1)			
		return 1			
	now finally	return 1 + 1 =2			0 1 1 2
i=4	fibonacci(4)	return fibonacci(3) + fibonacci(2)			
	taking first call	return fibonacci(2) + fibonacci(1)			
		return fibonacci(1) + fibonacci(0)			
		return 1			
		fibonacci(1) return 1			
		return 1 + 1 = 2			
	taking second call	fibonacci(1) + fibonacci(0)			
		return 1			
	finally	return 2 + 1 = 3			0 1 1 2 3
and so on...					

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

Example 5: Computing GCD

```
def gcd(a,b):
    if b==0:
        return a
    return gcd(b,a%b)

n1 = int(input("Enter First number "))
n2 = int(input("Enter Second number "))
r = gcd(n1,n2)
print("GCD of ",n1 ," and ", n2," is :",r)
```

```
Enter First number 15
Enter Second number 30
GCD of  15  and  30  is : 15
```

1. gcd(15,30)
   return (30,15)
2. gcd(30,15)
   return (15,0)
3. gcd(15,0)
   now b is 0, so return a
   i.e. 15

Answer: 15

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

Binary Searching

- Searching is a mechanism by which we look for any item in the list and give the result whether the item is present in the list or not.
- Searching is of basically 2 types:
  - Linear
  - Binary
- Linear searching is already covered in class XI, we will see how binary searching with and without recursion.

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

Binary Searching

- In Binary Searching the list to be search must be in sorted order i.e. either in ASCENDING ORDER or DESCENDING ORDER.
- In binary searching, we start comparison from middle element, if the middle element is equal to search item then search is over and element found is printed. If the middle element is larger than the search value then searching continues towards left side and otherwise towards right side (considering list is in ascending order)
- Middle position = (lower index + higher index) / 2

Example 6: Binary Searching(Iteration)

```
def BSearch(mylist,item):
    low = 0
    high = len(mylist)-1
    while(low<=high):
        mid = (low+high)//2
        if mylist[mid]==item:
            return mid
        elif mylist[mid]>item:
            high = mid - 1
        else:
            low = mid + 1
    return -1

list1 = [2,6,9,11,21,25,29,35,55,78,99]
n = int(input("Enter item to search :"))
pos = BSearch(list1,n)
if pos==-1:
    print("Not Found ")
else:
    print("Found @ position : ",pos+1)
```

```
Enter item to search :21
Found @ position : 5

(base) C:\Users\vin>python bsearch.py
Enter item to search :35
Found @ position : 8

(base) C:\Users\vin>python bsearch.py
Enter item to search :77
Not Found
```

Example 7: Binary Searching(Recursion)

```
def BSearch(mylist,item,low,high):
    if low>high:
        return -1

    mid = int((low+high)/2)
    if mylist[mid]==item:
        return mid
    elif mylist[mid]>item:
        high = mid - 1
        return BSearch(mylist,item,low,high)
    else:
        low = mid + 1
        return BSearch(mylist,item,low ,high)

list1 = [2,6,9,11,21,25,29,35,55,78,99]
n = int(input("Enter item to search :"))
pos = BSearch(list1,n,0,len(list1)-1)
if pos>=0:
    print("Found @ position ",pos+1)
else:
    print("Not Found")
```

```
Enter item to search :21
Found @ position : 5

(base) C:\Users\vin>python bsearch.py
Enter item to search :35
Found @ position : 8

(base) C:\Users\vin>python bsearch.py
Enter item to search :77
Not Found
```

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

Recursion Vs Iteration

BASIC FOR COMPARISON	RECURSION	ITERATION
BASIC	STATEMENT IN THE BODY OF FUNCTION CALLS THE FUNCTION ITSELF	ALLOWS SET OF INSTRUCTION TO BE EXECUTED REPEATEDLY
SPEED	SLOW IN EXECUTION	FAST IN EXECUTION
SIZE OF CODE	RECURSION REDUCES SIZE OF CODE	ITERATION MAKES THE CODE LONGER
APPLIED	ALWAYS APPLIED TO FUNCTION	APPLIED TO SET OF STATEMENTS
STACK USAGE	IT USES STACK	DOES NOT USE STACK
INFINITE	MAY CRASH THE SYSTEM	USES CPU CYCLE REPEATEDLY
OVERHEADS	IT POSSESSES THE OVERHEADS OF REPEATED FUNCTION CALL	NO OVERHEADS OF REPEATED FUNCTION CALL
CONDITION	MUST CONTAINS BASE CONDITION TO END THE RECURSION	MUST CONTAIN CONTROL CONDITIO TO END THE ITERATION