UNIT-5

Python Recursion: A function is said to be a recursive if it calls itself. For example, lets say we have a function abc() and in the body of abc() there is a call to the abc()

Python Example of Recursion:

In this example we are defining a user-defined function factorial(). This function finds the factorial of a number by calling itself repeadely until the base case

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

num = 5
print("Factorial of", num, "is:", factorial(num))
```

Output: Factorial of 5 is: 120

Explanation:

factorial(5) returns 5 * factorial(5-1)

i.e 5 * factorial(4)

↳ 5*4*factorial(3)

↳ 5*4*3*factorial(2)

↳ 5*4*3*2*factorial(1)

NOTE: factorial(1) is a base case for which we already know the value of factorial.

What is a base Case in Recursion?

When working with recursion, we should define a base case for which we already know the answer. In the above example we are finding factorial of an integer number and we already know that the factorial of 1 is 1 so this is our base case.

Each successive recursive call to the function should bring it closer to the base case, which is exactly what we are doing in above example.

We use base case in recursive function so that the function stops calling itself when the base case is reached. Without the base case, the function would keep calling itself indefinitely.

Why use Recursion in Programming?

We use recursion to break a big problem in small problems and those small problems into further smaller problems and so on. At the end the solutions of all the smaller subproblems are collectively helps in finding the solution of the big main problem.

Advantages:

Recursion makes our program:

1. Easier to write
2. Readable - code is easier to read and understand
3. Reduce the lines of code - It takes less lines of code to solve a problem using recursion.

Disadvantages:

1. Not all problems can be solved using recursion.
2. If you don't define the base case then the code would run indefinitely.
3. Debugging is difficult in recursive functions as the function is calling itself in a loop and it is hard to understand which call is causing the issue.
4. Memory overhead - Call to the recursive function is not memory efficient.

- Fibonacci Sequence Concept: The Fibonacci Sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 . . .

The next number is found by adding up the two numbers before it.

- The 2 is found by adding the two numbers before it (1+1).

- The 3 is found by adding the two numbers before it (1+2),
- And the 5 is (2+3),
- and so on!

Recursive Program for fibonacci Sequence:

$$F_n = F_{n-1} + F_{n+2}$$

where $F_0 = 0$ $F_1 = 1$

```
def Fibonacci(n):
    if n<0:
        print("Incorrect input")
    elif n==1:
        return 0
    elif n==2:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)

print(Fibonacci(9))
```

9-1 + 9-2

8 + 7

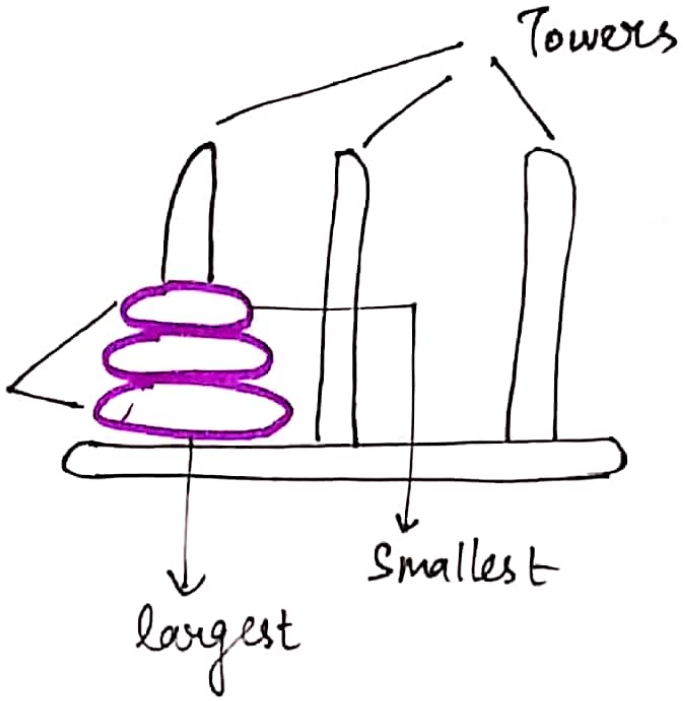
7 1 6 5

Output: 21

0, 1, 2

TONER OF HANOI

Tower of hanoi, is a mathematical puzzle which consists of three towers and more than one rings:



These rings are of diffrent sizes and stacked upon in an ascending order, i.e the smaller one sits over the larger one. There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

RULES: The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are -

- only one desk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can't sit over a small disk.

Recursive Program for Tower of Hanoi:

```
def TowerofHanoi(n, from_rod, to_rod, aux_rod):
    if n==1:
        print("Move disk 1 from Rod", from_rod,
        "to rod", to_rod)
        return
    TowerofHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk", n, "from rod", from_rod,
    "to rod", to_rod)
    TowerofHanoi(n-1, aux_rod, from_rod)

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

Output:

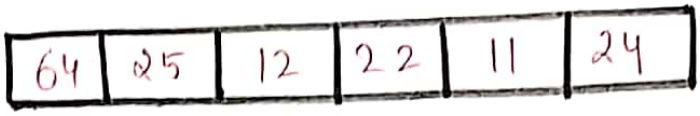
```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk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

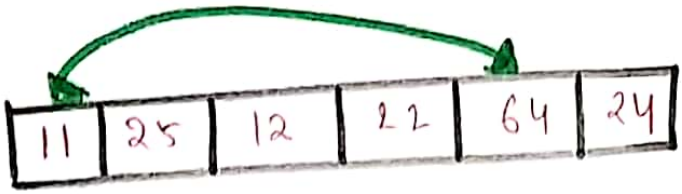
SELECTION SORT

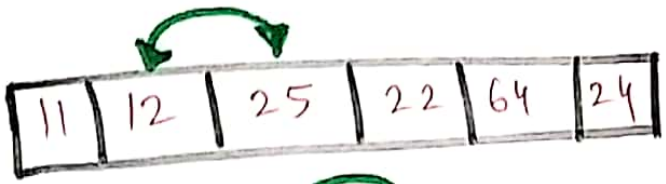
In Selection sort algorithm, an array is sorted by recursively finding the minimum element from the unsorted part and inserting it at the begining. Two subarrays are formed during the execution of Selection sort on a given array.

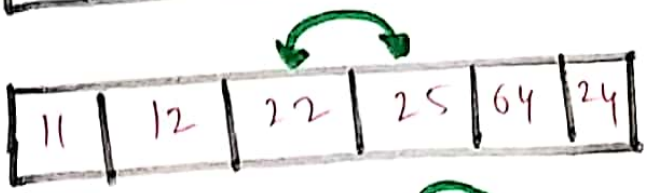
- The subarray, which is already sorted.
- The subarray, which is unsorted.

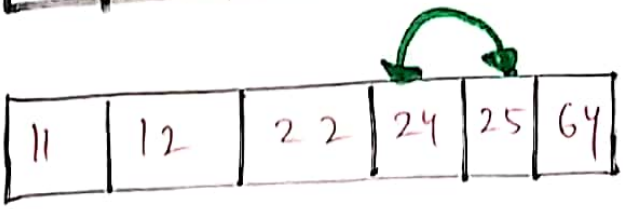
Let us consider an example to understand it clearly:



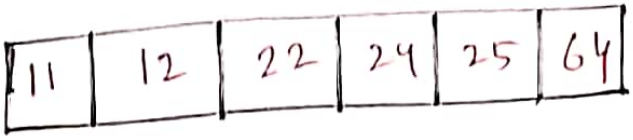








Result:



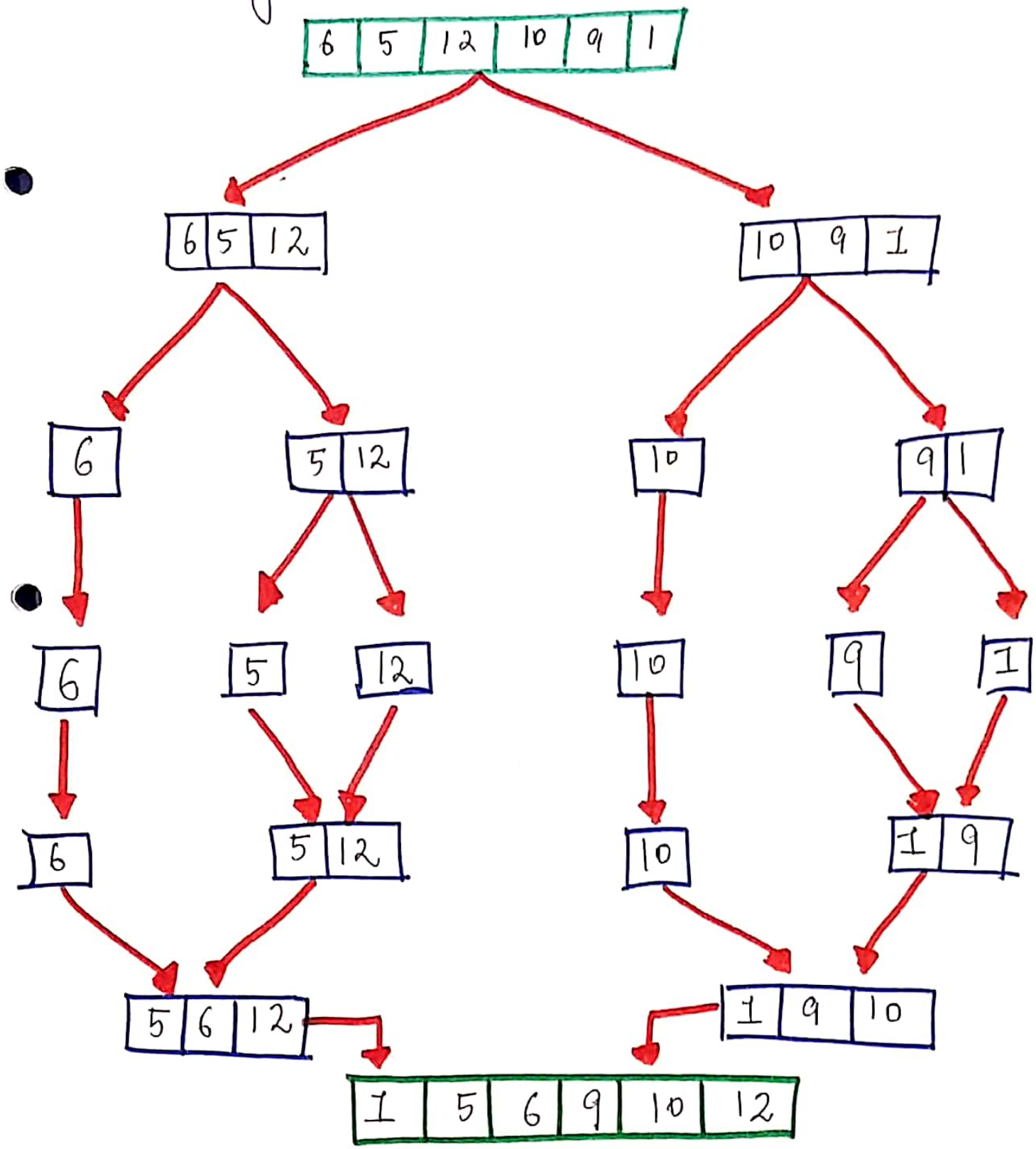
Program for Selection Sort:

```
import sys
A = [64, 25, 12, 22, 11]
for i in range (len(A)):
    min_idx = i
    for j in range (i+1, len(A)):
        if A [min_idx] > A [j]:
            min_idx = j
    A[i], A[min_idx] = A [min_idx], A [i]
print("Sorted array")
for i in range (len(A)):
    print ("%d" %A [i])
```

Output: Sorted Array

11 12 22 25 64

MERGE SORT: Merge sort is a kind of Divide and Conquer algorithm in computer programming. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.



Program in Python for Merge Sort:

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]
    i = 0
    j = 0
    k = l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
```

```
while j< n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort (arr, l, r):
        if l < r:
            m = (l+(r-1))/2
            mergeSort (arr, l, m)
            mergeSort (arr, m+1, r)
            merge (arr, l, m, r)

arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print ("Given array is")
for i in range (n):
        print ("%d" %arr[i])
mergesort (arr, 0, n-1)
print ("\n\nSorted array is")
for i in range (n):
        print ("%d" %arr[i])
```

Output: Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13

RECURSION AND ITERATION

Recursion and iteration both repeatedly executes the set of instructions. Recursion is when a statement in a function calls itself repeatedly. The iteration is when a loop repeatedly executes until the controlling condition becomes false.

The primary difference between recursion and iteration is that is a recursion is process, always applied to a function. The iteration is applied to the set of instructions which we want to get repeatedly executed.

Comparison Chart:

Basis for comparision	Recursion	Iteration
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.

Basis for comparisions	Recursion	Iteration
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution	Fast in execution
Size of code	Recursion reduces the size of the code.	Iteration make the code longer.

Key differences Between Recursion and iteration

1. Recursion is when a method in a program repeatedly calls itself whereas, iteration is when a set of instructions in a program are repeatedly executed.

2. A recursive method contains a set of instructions statement calling itself, and a termination condition whereas iteration statements contain initialization, increment, condition, set of instruction within a loop and a control variable.

3. A conditional statement decides the termination of recursion and control variable's value decide the termination of the iteration statement.

4. If the method does not lead to the termination condition it enters to infinite recursion. On the other hand, if the control variable never leads to the termination value the iteration statement iterates infinitely.