Practical 1

Aim: Write an algorithm and draw a flowchart to find the factorial of a given number.

Code:

```
num = int(input("Enter a number: "))
fact = 1

for i in range(1, num + 1):
    fact = fact * i

print("Factorial:", fact)
```

Practical 2

Aim: Write an algorithm and draw a flowchart to find the Fibonacci sequence.

Code:

```
n = int(input("Enter a number: "))
a = 0
b = 1

for i in range(n):
    print(a)
    c = a + b
    a = b
    b = c
```

Practical 3

Aim: Write an algorithm and draw a flowchart to find the GCD of a two number.

Code:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

while b != 0:
    a, b = b, a % b

print("GCD is:", a)
```

Practical 4

Aim: Write an algorithm and draw a flowchart to find the addition of the matrix.

Code:

```
 A = [[8,5,3],[4,4,6],[7,8,9]]
B = [[9,5,7],[6,8,4],[3,8,1]]
result = [[0,0,0],[0,0,0],[0,0,0]]

for i in range(len(A)):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] + B[i][j]

print("Resultant Matrix after addition:")
for row in result:
    print(row)
```

Practical 5

Aim: Write an algorithm and draw a flowchart to find the Multiplication of the matrix.

Code:

```
A = [[8,5,3],[3,4,6],[5,8,1]]
B = [[9,5,6],[6,2,4],[3,8,1]]
result = [[0,0,0],[0,0,0],[0,0,0]]

for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]

print("Multiplication:")
for row in result:
    print(row)
```

Practical 7

Aim: Write an algorithm and draw a flowchart to implement singly linked list.

Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Singly Linked List Class
class SinglyLinkedList:
```

```
def __init__(self):
    self.head = None

def insert(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
    else:
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

def delete(self, key):
    temp = self.head
    if temp and temp.data == key:
        self.head = temp.next
        return
    prev = None
    while temp and temp.data != key:
        prev = temp
        temp = temp.next
    if temp:
        prev.next = temp.next

def display(self):
    temp = self.head
    while temp:
        print(temp.data, end=" -> ")
```

```
            temp = temp.next
        print("None")

# Example
sll = SinglyLinkedList()
sll.insert(10)
sll.insert(20)
sll.insert(30)
print("List after insertion:")
sll.display()
sll.delete(20)
print("List after deleting 20:")
sll.display()
```

Practical 8

Aim: Write an algorithm and draw a flowchart to implement doubly linked list.

Code:

```
# Node class for Doubly Linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

# Doubly Linked List Class
class DoublyLinkedList:
    def __init__(self):
        self.head = None
    self.tail = None

# insert at beginning
def insert_at_beginning(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = self.tail = new_node
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node

# insert at end
def insert_at_end(self, data):
    new_node = Node(data)
    if self.tail is None:
        self.head = self.tail = new_node
    else:
        self.tail.next = new_node
        new_node.prev = self.tail
        self.tail = new_node

# delete from beginning
def delete_from_beginning(self):
    if self.head is None:
        print("List is empty , cannot delete.")
        return
    if self.head == self.tail:
        self.head = self.tail = None
    else:
        self.head = self.head.next
        self.head.prev = None

# delete from end
def delete_from_end(self):
    if self.tail is None:
        print("List is empty,cannot delete.")
        return
    if self.head == self.tail:
        self.head = self.tail = None
    else:
        self.tail = self.tail.prev
        self.tail.next = None

# display forward
def display_forward(self):
    current = self.head
    print("List (forward):", end=" ")
    while current:
        print(current.data, end="<->")
        current = current.next
    print("None")

# display backward
def display_backward(self):
    current = self.tail
    print("List (backward):", end=" ")
    while current:
            print(current.data, end="<->")
            current = current.prev
        print("None")

# example usage
d11 = DoublyLinkedList()
d11.insert_at_beginning(10)
d11.insert_at_end(20)
d11.insert_at_beginning(5)
d11.insert_at_end(30)
d11.display_forward()
d11.display_backward()
d11.delete_from_beginning()
d11.delete_from_end()
d11.display_forward()
d11.display_backward()
```

Practical 9

Aim: Write an algorithm and draw a flowchart to implement Circular linked list.

Code:

```
# Node class for circular Linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# circular Linked List Class
```

```
class CircularLinkedList:
    def __init__(self):
        self.head = None

    # function to insert a new node at the end of circular list
    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            new_node.next = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head

    # function to traverse and display the list
    def traverse(self):
        if not self.head:
            print("The circular linked list is empty.")
            return
        temp = self.head
        print("Circular Linked List Traversal:", end=" ")
        while True:
            print(temp.data, end=" ")
            temp = temp.next
            if temp == self.head:
                break
        print()

# Create a circular linked list
c11 = CircularLinkedList()
# insert elements into the list
c11.insert(10)
c11.insert(20)
c11.insert(5)
c11.insert(30)
# traverse and display elements of circular list
c11.traverse()
```

Practical 10

Aim: Write an algorithm and draw a flowchart to implement stack using array.

Code:

```
class StackArray:
    def __init__(self):
        # Initialize an empty list to represent the stack
        self.stack = []

    # Method to push (insert) element onto the stack
    def push(self, data):
        # Append element at the end (top of stack)
        self.stack.append(data)

    # Method to pop (remove) the top element from the stack
    def pop(self):
        # Check if stack is empty before popping
        if self.is_empty():
            print("Stack Underflow! Cannot pop.")  # Error message
            return None
        # Remove and return the last element
        return self.stack.pop()

    # Method to see the top element without removing it
    def peek(self):
        # If stack is empty, print message
        if self.is_empty():
            print("Stack is empty.")
            return None
        # Return the last element (top of stack)
        return self.stack[-1]

    # Method to check if stack is empty
    def is_empty(self):
        # Returns True if length is 0, else False
        return len(self.stack) == 0

    # Method to display stack elements
    def display(self):
        # Print stack from top to bottom by reversing list
        print("Stack (top -> bottom):", self.stack[::-1])

# Example usage of Stack
print("=== Stack using Array ===")  # Heading
s1 = StackArray()  # Create an object of StackArray
```

```
s1.push(10)  # Push 10 onto stack
s1.push(20)  # Push 20 onto stack
s1.push(30)  # Push 30 onto stack
s1.display()  # Display current stack
print("Popped:", s1.pop())  # Pop the top element (30)
s1.display()  # Display stack after popping
```

Practical 11

Aim: Write an algorithm and draw a flowchart to implement stack using array.

Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class StackLinkedList:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.is_empty():
            print("Stack Underflow! Cannot pop.")
            return None
        popped_data = self.top.data
        self.top = self.top.next
        return popped_data

    def peek(self):
        if self.is_empty():
            print("Stack is Empty.")
            return None
        return self.top.data

    def is_empty(self):
        return self.top is None

    def display(self):
        current = self.top
        print("Stack (top -> bottom):", end=" ")
        while current:
            print(current.data, end="->")
            current = current.next
        print("None")

print("\n===Stack using Linked List ===")
s2 = StackLinkedList()
s2.push(100)
s2.push(200)
s2.push(300)
s2.display()
```

```
print("Popped:", s2.pop())
s2.display()
```

Practical 12

Aim: Write an algorithm and flowchart to implement tower of Hanoi.

Code:

```
def hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return

    hanoi(n - 1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    hanoi(n - 1, auxiliary, target, source)

hanoi(3, 'A', 'C', 'B')
```

Practical 13

Aim:Write an algorithm and flowchart to implement Queue through array.

Code:

```
class QueueArray:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)
    def dequeue(self):
        if self.is_empty():
            return "Queue is empty!"
        return self.queue.pop(0)

    def peek(self):
        if self.is_empty():
            return "Queue is empty!"
        return self.queue[0]

    def is_empty(self):
        return len(self.queue) == 0

    def display(self):
        print("Queue:", self.queue)

print("Queue using Array")
qa = QueueArray()
qa.enqueue(10)
qa.enqueue(20)
qa.enqueue(30)
qa.display()
print("Dequeued:", qa.dequeue())
```

Practical 14

Aim:Write an algorithm and flowchart to implement Queue through linklist.

Code:

```
class Node:
    def __init__(self, data):
        self.data = data  # store the data
        self.next = None  # pointer to next node

class QueueLinkedList:
    def __init__(self):
        # Initialize empty queue with no front or rear
        self.front = self.rear = None

    def enqueue(self, item):
        """Insert item at the rear of the queue"""
        new_node = Node(item)  # create a new node
        if self.rear is None:  # if queue is empty
            self.front = self.rear = new_node
            return
        # link new node at the end of the queue
        self.rear.next = new_node
        # make new node the rear
        self.rear = new_node

    def dequeue(self):
        """Remove and return the front item"""
        if self.front is None:
            return "Queue is empty!"
        temp = self.front
        # move front to the next node
        self.front = temp.next

        # if queue becomes empty, set rear to None
        if self.front is None:
            self.rear = None
        return temp.data

    def peek(self):
        """Return the front element without removing"""
        if self.front is None:
            return "Queue is empty!"
        return self.front.data

    def is_empty(self):
        """Check if queue is empty"""
        return self.front is None

    def display(self):
        """Traverse and display the queue"""
        if self.front is None:
            print("Queue is empty!")
            return
        temp = self.front
        while temp:
            print(temp.data, end=" <- ")
            temp = temp.next
        print("None")

# Example usage
print("Queue using Linked List")
ql = QueueLinkedList()
ql.enqueue(10)  # Add 10
ql.enqueue(20)  # Add 20
ql.enqueue(30)  # Add 30
ql.display()    # 10 <- 20 <- 30 <- None
print("Dequeued:", ql.dequeue())  # Removes 10
ql.display()    # 20 <- 30 <- None
```

Practical 15

Aim – Implement priority queue using array or linked list.

Code –

```
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def insert(self, element, priority):
        self.queue.append((element, priority))

    def delete(self):
        if not self.queue:
            print("Queue is empty!")
            return

        # Find element with highest priority (lowest number = highest priority)
        min_priority_index = 0
        for i in range(len(self.queue)):
            if self.queue[i][1] < self.queue[min_priority_index][1]:
                min_priority_index = i

        element = self.queue.pop(min_priority_index)
        print(f"Deleted element: {element[0]} (priority={element[1]})")
    def display(self):
        if not self.queue:
            print("Queue is empty!")
        else:
            print("Priority Queue:")
            for item in self.queue:
                print(f"{item[0]} (priority={item[1]})")

# Example usage
pq = PriorityQueue()
pq.insert("A", 3)
pq.insert("B", 1)
pq.insert("C", 2)
pq.display()
pq.delete()
pq.display()
```

Practical 16

Aim - Implement binary tree and perform inorder,preorder,postorder traversal.

Code –

```
class node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

```
class binarytree:
    def __init__(self):
        self.root = None

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.data, end='')
            self.inorder(node.right)

    def preorder(self, node):
        if node:
            print(node.data, end='')
            self.preorder(node.left)
            self.preorder(node.right)

    def postorder(self, node):
        if node:
            self.postorder(node.left)
            self.postorder(node.right)
            print(node.data, end='')

tree = binarytree()
tree.root = node(1)
tree.root.left = node(2)
tree.root.right = node(3)
tree.root.left.left = node(4)
tree.root.left.right = node(5)
```

```
print("inorder traversal")
tree.inorder(tree.root)
print("\npreorder traversal")
tree.preorder(tree.root)
print("\npostorder traversal")
tree.postorder(tree.root)
```

Practical 17

Aim - Implement Binary Search tree and perform insert ,delete , and search operations

Code –

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    # Insert operation
    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
```

```
        root.right = self.insert(root.right, key)
    return root

# Search operation
def search(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self.search(root.left, key)
    return self.search(root.right, key)

# Delete operation
def delete(self, root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = self.delete(root.left, key)
    elif key > root.key:
        root.right = self.delete(root.right, key)
    else:
        # Node with one or no child
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        # Node with two children — get inorder successor
        min_node = self.get_min_value_node(root.right)
        root.key = min_node.key
```

```
            root.right = self.delete(root.right, min_node.key)
        return root

    # Get the smallest value in the tree
    def get_min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    # Inorder traversal (sorted order)
    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)

if __name__ == "__main__":
    tree = BST()
    root = None

    # Insert nodes
    for key in [50, 34, 25, 78, 11, 90, 49]:
        root = tree.insert(root, key)

    print("Inorder traversal (should be sorted):")
    tree.inorder(root)
    print("\n")
    # Search a key
    key_to_search = int(input("Enter key to search: "))
    result = tree.search(root, key_to_search)
    if result:
        print(f"Key {key_to_search} found in the BST.")
    else:
        print(f"Key {key_to_search} not found in the BST.")

    # Delete a key
    key_to_delete = int(input("\nEnter key to delete: "))
    root = tree.delete(root, key_to_delete)
    print(f"\nAfter deleting {key_to_delete}, inorder traversal:")
    tree.inorder(root)
    print()
```

Practical 18

Aim - Implement AVL tree and demonstrate rotations

Code –

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def get_height(self, root):
        if not root:
```

```
        return 0
    return root.height

def get_balance(self, root):
    if not root:
        return 0
    return self.get_height(root.left) - self.get_height(root.right)

def right_rotate(self, y):
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
    x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))

    return x

def left_rotate(self, x):
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
    return y

def insert(self, root, key):
    if not root:
        return Node(key)
    elif key < root.key:
        root.left = self.insert(root.left, key)
    elif key > root.key:
        root.right = self.insert(root.right, key)
    else:
        return root

    root.height = 1 + max(self.get_height(root.left),
                          self.get_height(root.right))

    balance = self.get_balance(root)

    if balance > 1 and key < root.left.key:
        return self.right_rotate(root)

    if balance < -1 and key > root.right.key:
        return self.left_rotate(root)

    if balance > 1 and key > root.left.key:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    if balance < -1 and key < root.right.key:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root

def get_min_value_node(self, root):
    if root is None or root.left is None:
        return root
    return self.get_min_value_node(root.left)

def delete(self, root, key):
    if not root:
        return root
    elif key < root.key:
        root.left = self.delete(root.left, key)
    elif key > root.key:
        root.right = self.delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        temp = self.get_min_value_node(root.right)
        root.key = temp.key
        root.right = self.delete(root.right, temp.key)

    if root is None:
        return root
    root.height = 1 + max(self.get_height(root.left),
                          self.get_height(root.right))

    balance = self.get_balance(root)

    if balance > 1 and self.get_balance(root.left) >= 0:
        return self.right_rotate(root)

    if balance > 1 and self.get_balance(root.left) < 0:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    if balance < -1 and self.get_balance(root.right) <= 0:
        return self.left_rotate(root)

    if balance < -1 and self.get_balance(root.right) > 0:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root

def inorder(self, root):
    if not root:
        return
    self.inorder(root.left)
    print(root.key, end=" ")
    self.inorder(root.right)
```

```
if __name__ == "__main__":
    tree = AVLTree()
    root = None

    nums = [10, 20, 30, 40, 50, 25]
    for num in nums:
        root = tree.insert(root, num)

    print("Inorder traversal after insertions:")
    tree.inorder(root)
    print()

    root = tree.delete(root, 40)
    print("Inorder traversal after deleting 40:")
    tree.inorder(root)
    print()
```

Practical 20

Aim: Write an algorithm and draw a flowchart to implement BFS and DFS.

Code:

```
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    # Breadth First Search
    def bfs(self, start):
        visited = set()
        queue = deque([start])
        visited.add(start)
        print("BFS Traversal:", end="")
        while queue:
            vertex = queue.popleft()
            print(vertex, end=" ")
            for neighbor in self.graph[vertex]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        print()

    # Depth First Search
    def dfs(self, start):
        visited = set()
        print("DFS Traversal:", end=" ")
        self._dfs_recursive(start, visited)
        print()

    def _dfs_recursive(self, vertex, visited):
        visited.add(vertex)
        print(vertex, end=" ")
        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                self._dfs_recursive(neighbor, visited)


if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(3, 6)

    print("Adjacency List Representation:")
    for vertex in g.graph:
        print(f"{vertex} -> {g.graph[vertex]}")
    print()

    g.bfs(0)
    g.dfs(0)
```

Practical 20

Aim: Write an algorithm and draw a flowchart to implement BFS and DFS.

Code:

```
import sys  # For using system max size as infinite


class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.graph = [[0 for _ in range(vertices)] for _ in range(vertices)]  # Adjacency
matrix

    # Function to print the constructed MST
    def print_mst(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(f"{parent[i]} - {i} \t{self.graph[i][parent[i]]}")

    # Function to find the vertex with the minimum key value
    def min_key(self, key, mst_set):
        min_val = sys.maxsize
        min_index = -1
        for v in range(self.V):
            if key[v] < min_val and not mst_set[v]:
                min_val = key[v]
                min_index = v
        return min_index

    # Function to construct MST using Prim's algorithm
    def prim_mst(self):
        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = 0
        mst_set = [False] * self.V
        parent[0] = -1
        for _ in range(self.V):
            u = self.min_key(key, mst_set)
            mst_set[u] = True

            for v in range(self.V):
                if self.graph[u][v] > 0 and not mst_set[v] and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.print_mst(parent)

# ----------------- Main Program -----------------
if __name__ == "__main__":
    g = Graph(5)
    g.graph = [
        [0, 2, 0, 6, 0],
        [2, 0, 3, 8, 5],
        [0, 3, 0, 0, 7],
        [6, 8, 0, 0, 9],
        [0, 5, 7, 9, 0]
    ]
    g.prim_mst()
```

Practical 21

Aim: Implement Dijkstra’s algorithm for finding shortest path.

Code:

```
import sys  # Import sys module to use sys.maxsize (represents infinity)

# Define a class to represent the graph
```

```
class Graph:
    def __init__(self, vertices):
        # Initialize number of vertices in the graph
        self.V = vertices
        # Create a 2D list (adjacency matrix) to store graph weights
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    # Function to print the final shortest distances from source to each vertex
    def print_solution(self, dist):
        print("Vertex \t At distance from Source")
        for node in range(self.V):
            print(node, "\t\t", dist[node])

    # Function to find the vertex with the minimum distance value
    # from the set of vertices not yet processed
    def min_distance(self, dist, spt_set):
        min_val = sys.maxsize
        min_index = -1

        for v in range(self.V):
            if dist[v] < min_val and spt_set[v] == False:
                min_val = dist[v]
                min_index = v
        return min_index

    # Function that implements Dijkstra's shortest path algorithm
    def dijkstra(self, src):
        dist = [sys.maxsize] * self.V
        dist[src] = 0
        spt_set = [False] * self.V

        for _ in range(self.V):
            u = self.min_distance(dist, spt_set)
            spt_set[u] = True

            for v in range(self.V):
                if (self.graph[u][v] > 0 and
                    spt_set[v] == False and
                    dist[v] > dist[u] + self.graph[u][v]):
                    dist[v] = dist[u] + self.graph[u][v]

        self.print_solution(dist)

# DRIVER CODE
g = Graph(9)

g.graph = [
    [0, 4, 0, 0, 0, 0, 0, 8, 0],
    [4, 0, 8, 0, 0, 0, 0, 11, 0],
    [0, 8, 0, 7, 0, 4, 0, 0, 2],
    [0, 0, 7, 0, 9, 14, 0, 0, 0],
    [0, 0, 0, 9, 0, 10, 0, 0, 0],
    [0, 0, 4, 14, 10, 0, 2, 0, 0],
    [0, 0, 0, 0, 0, 2, 0, 1, 6],
    [8, 11, 0, 0, 0, 0, 1, 0, 7],
    [0, 0, 2, 0, 0, 0, 6, 7, 0]
]
```

```
g.dijkstra(0)
```

Practical 22

Aim: Implement Bubble Sort.

Code:

```
def bubble_sort(arr):
    """
    Sorts a list of comparable elements using the Bubble Sort algorithm.
    Args:
        arr (list): The list to be sorted.
    """
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        # Flag to optimize the sort: if no two elements were swapped
        swapped = False

        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Compare the adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap them if they are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # If no two elements were swapped, array is sorted
        if not swapped:
            break

# --- Example Usage ---
my_list = [40, 20, 75, 2, 55, 10, 5]
print(f"Original list: {my_list}")
bubble_sort(my_list)
print(f"Sorted list: {my_list}")
```

Practical No.-23

Aim: - Write a python program to perform Selection Sort.

Code:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Driver code
arr = [64, 25, 12, 22, 11]
print("Original array:", arr)
selection_sort(arr)
print("Sorted array:", arr)
```

Practical No.-24

Aim: - Write a python program to perform Insertion Sort.

Code:

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1

        arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
print("Original array:", arr)
insertionSort(arr)
print("Sorted array:", arr)
```

Practical No.-25

Aim: - Write a python program to perform bubble Sort.

Code:

```
def bubbleSort(arr):
    n = len(arr)

    for i in range(n - 1):
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print("Original array:", arr)
bubbleSort(arr)
print("Sorted array:", arr)
```

Practical No.-26

Aim: - Write a python program to perform Merge Sort.

Code:

```
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

def mergeSort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = mergeSort(arr[:mid])
    right = mergeSort(arr[mid:])

    return merge(left, right)

arr = [38, 27, 43, 10]

sorted_arr = mergeSort(arr)
print("Original array:", arr)
print("Sorted array:", sorted_arr)
```

Practical No.-27

Aim: - Write a python program to perform Heap Sort.

Code:

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

arr = [9, 4, 3, 8, 10, 2, 5]
print("Original array:", arr)
heapSort(arr)
print("Sorted array:", arr)
```

PRACTICAL 28

Aim:- Implement Binary Search

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

arr = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
target = int(input("Enter the number to search: "))

result = binary_search(arr, target)

if result != -1:
    print(f"Found at index: {result}")
else:
    print("Not found in the array")
```

PRACTICAL 29

Aim:- Implement Linear Search

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

# Fixed array
arr = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

# Get target input from user
target = int(input("Enter the number to search: "))
```

```
# Perform linear search
result = linear_search(arr, target)

if result != -1:
    print(f"Found at index: {result}")
else:
    print("Not found in the array")
```

PRACTICAL 30

Aim:- Implement quick sort:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)

arr = [64, 34, 25, 12, 22, 11, 90, 88, 45, 50]

print("Original array:", arr)
sorted_arr = quick_sort(arr)
print("Sorted array:", sorted_arr)
```