1. Write a Python program for how to create and manipulate arrays.

Program code:

```
# Import array module
import array as arr

# Create an array of integers
numbers = arr.array('i', [10, 20, 30, 40, 50])

# Display the original array
print("Original array:", numbers)

# Access elements using index
print("First element:", numbers[0])
print("Last element:", numbers[-1])

# Insert a new element at the end
numbers.append(60)
print("After appending 60:", numbers)

# Insert element at a specific position
numbers.insert(2, 25)
print("After inserting 25 at index 2:", numbers)

# Remove an element
numbers.remove(40)
print("After removing 40:", numbers)

# Pop last element
numbers.pop()
print("After popping last element:", numbers)

# Update an element
numbers[1] = 15
print("After updating index 1 to 15:", numbers)

# Traverse the array
print("Traversing the array:")
for num in numbers:
    print(num, end=" ")

# Find length of array
print("\nLength of array:", len(numbers))
```

Output -

```
Original array: array('i', [10, 20, 30, 40, 50])
First element: 10
Last element: 50
After appending 60: array('i', [10, 20, 30, 40, 50, 60])
After inserting 25 at index 2: array('i', [10, 20, 25, 30, 40, 50, 60])
After removing 40: array('i', [10, 20, 25, 30, 50, 60])
After popping last element: array('i', [10, 20, 25, 30, 50])
After updating index 1 to 15: array('i', [10, 15, 25, 30, 50])
Traversing the array:
10 15 25 30 50
Length of array: 5
```

2. **Write a Python program to perform basic operations such as insertion, deletion, and traversal in 1D array**

Program code:

```
# Import array module
import array as arr

# Create an array of integers
numbers = arr.array('i', [10, 20, 30, 40, 50])
print("Original Array:", numbers)

# -------- Traversal --------
print("\nTraversal of array elements:")
for i in range(len(numbers)):
    print(f"Index {i} -> {numbers[i]}")

# -------- Insertion --------
# Insert element 60 at the end
numbers.append(60)
print("\nAfter appending 60:", numbers)

# Insert element 25 at index 2
numbers.insert(2, 25)
print("After inserting 25 at index 2:", numbers)

# -------- Deletion --------
# Remove element by value
```

```
numbers.remove(40)
print("\nAfter removing 40:", numbers)

# Delete last element using pop()
numbers.pop()
print("After popping last element:", numbers)

# -------- Traversal after operations --------
print("\nFinal array elements after all operations:")
for num in numbers:
    print(num, end=" ")

print("\nLength of final array:", len(numbers))
```

Output -

```
Original Array: array('i', [10, 20, 30, 40, 50])

Traversal of array elements:
Index 0 -> 10
Index 1 -> 20
Index 2 -> 30
Index 3 -> 40
Index 4 -> 50

After appending 60: array('i', [10, 20, 30, 40, 50, 60])
After inserting 25 at index 2: array('i', [10, 20, 25, 30, 40, 50, 60])

After removing 40: array('i', [10, 20, 25, 30, 50, 60])
After popping last element: array('i', [10, 20, 25, 30, 50])

Final array elements after all operations:
10 20 25 30 50
Length of final array: 5
```

3. **Write a Python program for implementing a function that searches for an element in a 1-D array and returns its index.**

Program code:

```
# Import array module
import array as arr

# Function to search for an element and return its index
def search_element(array, value):
    for i in range(len(array)):
        if array[i] == value:
            return i  # Return the index if element is found
    return -1  # Return -1 if element not found

# Create an array of integers
numbers = arr.array('i', [10, 20, 30, 40, 50, 60])
print("Array elements:", numbers)

# Input element to search
value = int(input("\nEnter the element to search: "))

# Function call
index = search_element(numbers, value)

# Display result
if index != -1:
    print(f"\nElement {value} found at index {index}.")
else:
    print(f"\nElement {value} not found in the array.")
```

Output –

```
Array elements: array('i', [10, 20, 30, 40, 50, 60])

Enter the element to search: 20

Element 20 found at index 1.
```

4. **Write a Python program for how to represent and manipulate sparse matrices.**

Program code:

```
# Create a sparse matrix
def create_sparse(matrix):
    sparse = {}
```

```
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] != 0:     # Only store non-zero values
                sparse[(i, j)] = matrix[i][j]
    return sparse

# Display sparse matrix
def display_sparse(sparse):
    print("\nSparse Matrix Representation (row, col): value")
    for key, value in sparse.items():
        print(key, ":", value)

# Add two sparse matrices
def add_sparse(s1, s2):
    result = s1.copy()
    for key, value in s2.items():
        if key in result:
            result[key] += value
            if result[key] == 0:
                del result[key]     # Remove zero entries
        else:
            result[key] = value
    return result

# Multiply sparse matrix by scalar
def scalar_multiply(sparse, k):
    result = {}
    for key, value in sparse.items():
        result[key] = value * k
    return result


# ------------ MAIN PROGRAM ------------

# Dense matrix (example)
matrix = [
    [0, 0, 3, 0],
    [4, 0, 0, 0],
    [0, 5, 0, 6]
]

print("Original Matrix (Dense):")
for row in matrix:
```

```
    print(row)

# Create sparse matrix
sparse_matrix = create_sparse(matrix)
display_sparse(sparse_matrix)

# Add sparse matrix to itself
added_matrix = add_sparse(sparse_matrix, sparse_matrix)
print("\nAfter Adding Sparse Matrix to Itself:")
display_sparse(added_matrix)

# Multiply sparse matrix by scalar
scalar_result = scalar_multiply(sparse_matrix, 3)
print("\nAfter Scalar Multiplication by 3:")
display_sparse(scalar_result)
```

Output –

```
Original Matrix (Dense):
[0, 0, 3, 0]
[4, 0, 0, 0]
[0, 5, 0, 6]

Sparse Matrix Representation (row, col): value
(0, 2) : 3
(1, 0) : 4
(2, 1) : 5
(2, 3) : 6

After Adding Sparse Matrix to Itself:

Sparse Matrix Representation (row, col): value
(0, 2) : 6
(1, 0) : 8
(2, 1) : 10
(2, 3) : 12

After Scalar Multiplication by 3:

Sparse Matrix Representation (row, col): value
(0, 2) : 9
(1, 0) : 12
(2, 1) : 15
(2, 3) : 18
```

5. **Write a Python function that reverses a string using an array.**
   Program code:

```
def reverse_string(text):
    # Convert string to an array (list of characters)
    char_array = list(text)

    # Reverse the array manually
    start = 0
    end = len(char_array) - 1

    while start < end:
        # Swap characters
        char_array[start], char_array[end] = char_array[end], char_array[start]
```

```
        start += 1
        end -= 1

    # Convert the array back to a string
    return ''.join(char_array)


# -------- MAIN PROGRAM --------
string_input = input("Enter a string to reverse: ")

reversed_string = reverse_string(string_input)

print("Reversed String:", reversed_string)
```

Output –

```
Enter a string to reverse: Hello
Reversed String: olleH
```

6. **Write a Python program for singly linked list to create a linked list and display it.**

Program code:

```
class Node:
    def __init__(self, data):
        self.data = data      # Store data
        self.next = None      # Pointer to next node


# Linked List class
class LinkedList:
    def __init__(self):
        self.head = None      # Initially empty list

    # Function to add a new node at the end
    def append(self, data):
        new_node = Node(data)

        # If list is empty
        if self.head is None:
            self.head = new_node
            return

        # Traverse to the last node
```

```
        temp = self.head
        while temp.next:
            temp = temp.next

        # Link new node
        temp.next = new_node

    # Function to display the linked list
    def display(self):
        if self.head is None:
            print("Linked List is empty")
            return

        print("Linked List elements:", end=" ")
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")


# -------- MAIN PROGRAM --------

# Create linked list object
ll = LinkedList()

# Add elements to the list
ll.append(10)
ll.append(20)
ll.append(30)
ll.append(40)

# Display the linked list
ll.display()
```

Output –

```
Linked List elements: 10 -> 20 -> 30 -> 40 -> None
```

7. **Write a Python program for singly linked list to search given data in the linked list.**

Program code:

```
class Node:
    def __init__(self, data):
```

```
        self.data = data
        self.next = None


# Linked List class
class LinkedList:
    def __init__(self):
        self.head = None

    # Function to add node at the end
    def append(self, data):
        new_node = Node(data)

        # If linked list is empty
        if self.head is None:
            self.head = new_node
            return

        temp = self.head
        while temp.next:
            temp = temp.next

        temp.next = new_node

    # Function to search for an element
    def search(self, key):
        temp = self.head
        position = 0

        while temp:
            if temp.data == key:
                return position     # Return index if found
            temp = temp.next
            position += 1

        return -1   # Not found

    # Function to display linked list
    def display(self):
        if self.head is None:
            print("Linked List is empty.")
            return
```

```
        print("Linked List:", end=" ")
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")


# -------- MAIN PROGRAM --------
ll = LinkedList()

# Adding elements to the linked list
ll.append(10)
ll.append(20)
ll.append(30)
ll.append(40)

ll.display()

# Input data to search
value = int(input("\nEnter value to search in linked list: "))

# Search operation
index = ll.search(value)

if index != -1:
    print(f"\nValue {value} found at position {index}.")
else:
    print(f"\nValue {value} NOT found in the linked list.")
```

Output –

```
Linked List: 10 -> 20 -> 30 -> 40 -> None

Enter value to search in linked list: 20

Value 20 found at position 1.
```

8. **Write a Python program for doubly linked list to insert a node at the last position of the linked list.**

Program code:

```
class Node:
    def __init__(self, data):
        self.data = data       # Store data
        self.prev = None       # Pointer to previous node
        self.next = None       # Pointer to next node


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

    # Function to insert node at the end of the list
    def insert_last(self, data):
        new_node = Node(data)

        # If list is empty
        if self.head is None:
            self.head = new_node
            return

        # Traverse to the last node
        temp = self.head
        while temp.next:
            temp = temp.next

        # Insert new node at the end
        temp.next = new_node
        new_node.prev = temp

    # Display list from start to end
    def display(self):
        if self.head is None:
            print("Doubly Linked List is empty.")
            return

        print("Doubly Linked List (forward):", end=" ")
        temp = self.head
        while temp:
            print(temp.data, end=" <-> ")
            temp = temp.next
        print("None")
```

```
# -------- MAIN PROGRAM --------

dll = DoublyLinkedList()

# Insert nodes at last
dll.insert_last(10)
dll.insert_last(20)
dll.insert_last(30)
dll.insert_last(40)

# Display the doubly linked list
dll.display()
```

Output –

```
Doubly Linked List (forward): 10 <-> 20 <-> 30 <-> 40 <-> None
```

9. **Write a Python program for doubly linked list to delete a node from a specified position.**

Program code:

```
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

    # Insert node at the end
    def insert_last(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return

        temp = self.head
        while temp.next:
            temp = temp.next
```

```
        temp.next = new_node
        new_node.prev = temp

    # Delete node at a given position (0-based index)
    def delete_at_position(self, pos):
        if self.head is None:
            print("List is empty.")
            return

        temp = self.head

        # Deleting the first node
        if pos == 0:
            self.head = temp.next
            if self.head:
                self.head.prev = None
            del temp
            return

        # Traverse to the node at position pos
        for i in range(pos):
            temp = temp.next
            if temp is None:
                print("Position out of range.")
                return

        # Adjust pointers
        if temp.next:
            temp.next.prev = temp.prev
        if temp.prev:
            temp.prev.next = temp.next

        del temp

    # Display list forward
    def display(self):
        if self.head is None:
            print("Doubly Linked List is empty.")
            return

        temp = self.head
        print("Doubly Linked List (forward):", end=" ")
```

```
        while temp:
            print(temp.data, end=" <-> ")
            temp = temp.next
        print("None")


# -------- MAIN PROGRAM --------

dll = DoublyLinkedList()

# Insert nodes
dll.insert_last(10)
dll.insert_last(20)
dll.insert_last(30)
dll.insert_last(40)
dll.insert_last(50)

print("Original List:")
dll.display()

# Input position to delete
pos = int(input("\nEnter the position of the node to delete (0-based index): "))

# Delete node at specified position
dll.delete_at_position(pos)

print("\nList after deletion:")
dll.display()
```

Output –

```
Original List:
Doubly Linked List (forward): 10 <-> 20 <-> 30 <-> 40 <-> 50 <-> None

Enter the position of the node to delete (0-based index): 2

List after deletion:
Doubly Linked List (forward): 10 <-> 20 <-> 40 <-> 50 <-> None
```

10. **Write a Python program to create a singly linked list, count the total number of nodes in it, and display the result.**

Program code:

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


# Singly Linked List class
class LinkedList:
    def __init__(self):
        self.head = None

    # Function to append node at the end
    def append(self, data):
        new_node = Node(data)

        # If list is empty
        if self.head is None:
            self.head = new_node
            return

        # Traverse to the last node
        temp = self.head
        while temp.next:
            temp = temp.next

        temp.next = new_node

    # Function to count total nodes
    def count_nodes(self):
        count = 0
        temp = self.head
        while temp:
            count += 1
            temp = temp.next
        return count

    # Function to display linked list
    def display(self):
        if self.head is None:
            print("Linked List is empty.")
            return

        temp = self.head
```

```
        print("Linked List:", end=" ")
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")


# -------- MAIN PROGRAM --------

# Create linked list object
ll = LinkedList()

# Add nodes
ll.append(10)
ll.append(20)
ll.append(30)
ll.append(40)

# Display the linked list
ll.display()

# Count total nodes
total_nodes = ll.count_nodes()
print(f"\nTotal number of nodes in the linked list: {total_nodes}")
```

Output –

```
Linked List: 10 -> 20 -> 30 -> 40 -> None

Total number of nodes in the linked list: 4
```

11. Write a Python program for doubly linked list to sort the linked list in ascending order.

Program code:

```
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

# Insert node at the end
def insert_last(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        return

    temp = self.head
    while temp.next:
        temp = temp.next

    temp.next = new_node
    new_node.prev = temp

# Display the list
def display(self):
    if self.head is None:
        print("Doubly Linked List is empty.")
        return

    temp = self.head
    print("Doubly Linked List:", end=" ")
    while temp:
        print(temp.data, end=" <-> ")
        temp = temp.next
    print("None")

# Sort the linked list in ascending order using Bubble Sort
def sort_ascending(self):
    if self.head is None:
        return

    swapped = True
    while swapped:
        swapped = False
        current = self.head
        while current.next:
            if current.data > current.next.data:
                # Swap data
                current.data, current.next.data = current.next.data, current.data
```

```
                swapped = True
            current = current.next


# -------- MAIN PROGRAM --------

dll = DoublyLinkedList()

# Insert nodes
dll.insert_last(40)
dll.insert_last(10)
dll.insert_last(30)
dll.insert_last(20)
dll.insert_last(50)

print("Original List:")
dll.display()

# Sort the list in ascending order
dll.sort_ascending()

print("\nSorted List in Ascending Order:")
dll.display()
```

Output –

```
Original List:
Doubly Linked List: 40 <-> 10 <-> 30 <-> 20 <-> 50 <-> None

Sorted List in Ascending Order:
Doubly Linked List: 10 <-> 20 <-> 30 <-> 40 <-> 50 <-> None
```

12. **Write a Python program for binary search tree creation and insertion of elements.**

Program code:

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

```
# Binary Search Tree class
class BST:
    def __init__(self):
        self.root = None

    # Function to insert a key
    def insert(self, key):
        self.root = self._insert_recursive(self.root, key)

    # Helper recursive function for insertion
    def _insert_recursive(self, root, key):
        # If tree/subtree is empty
        if root is None:
            return Node(key)

        # If key is smaller, go to left subtree
        if key < root.key:
            root.left = self._insert_recursive(root.left, key)
        # If key is greater, go to right subtree
        elif key > root.key:
            root.right = self._insert_recursive(root.right, key)
        # Duplicate keys are not allowed
        return root

    # In-order traversal (Left, Root, Right)
    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)


# -------- MAIN PROGRAM --------

bst = BST()

# Insert elements
elements = [50, 30, 20, 40, 70, 60, 80]
for el in elements:
    bst.insert(el)

# Display BST using in-order traversal
```

```
print("In-order Traversal of BST:")
bst.inorder(bst.root)
```

Output –

```
In-order Traversal of BST:
20 30 40 50 60 70 80
```

13. Write a Python program for implementing graphs with DFS traversal.

Program code:

```
class Graph:
    def __init__(self):
        self.graph = {}  # Dictionary to store adjacency list

    # Add edge (undirected 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)  # Remove this line for directed graph

    # DFS traversal starting from a given node
    def dfs(self, start):
        visited = set()  # To keep track of visited nodes
        self._dfs_recursive(start, visited)

    # Helper recursive function for DFS
    def _dfs_recursive(self, node, visited):
        visited.add(node)
        print(node, end=" ")

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self._dfs_recursive(neighbor, visited)


# -------- MAIN PROGRAM --------

g = Graph()
```

```
# Add edges
edges = [
    (0, 1),
    (0, 2),
    (1, 2),
    (2, 3),
    (3, 4),
]

for u, v in edges:
    g.add_edge(u, v)

# Display adjacency list
print("Graph Adjacency List:")
for node in g.graph:
    print(f"{node}: {g.graph[node]}")

# Perform DFS starting from node 0
print("\nDFS Traversal starting from node 0:")
g.dfs(0)
```

Output –

```
Graph Adjacency List:
0: [1, 2]
1: [0, 2]
2: [0, 1, 3]
3: [2, 4]
4: [3]

DFS Traversal starting from node 0:
0 1 2 3 4
```

14. Write a Python program for implementing graphs with BFS traversal.

Program code:

```
from collections import deque

# Graph class using adjacency list
class Graph:
    def __init__(self):
        self.graph = {}  # Dictionary to store adjacency list

    # Add edge (undirected 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)  # Remove this line for directed graph

    # BFS traversal starting from a given node
    def bfs(self, start):
        visited = set()
        queue = deque()

        visited.add(start)
        queue.append(start)

        while queue:
            node = queue.popleft()
            print(node, end=" ")

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)


# -------- MAIN PROGRAM --------

g = Graph()

# Add edges
edges = [
    (0, 1),
    (0, 2),
    (1, 2),
    (2, 3),
    (3, 4),
]

for u, v in edges:
    g.add_edge(u, v)
```

```
# Display adjacency list
print("Graph Adjacency List:")
for node in g.graph:
    print(f"{node}: {g.graph[node]}")

# Perform BFS starting from node 0
print("\nBFS Traversal starting from node 0:")
g.bfs(0)
```

Output –

```
Graph Adjacency List:
0: [1, 2]
1: [0, 2]
2: [0, 1, 3]
3: [2, 4]
4: [3]

BFS Traversal starting from node 0:
0 1 2 3 4
```

15. Write a Python Program for linear search.

Program code:

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index  # Return index if found
    return -1  # Return -1 if not found


# -------- MAIN PROGRAM --------

# Sample list
numbers = [10, 20, 30, 40, 50, 60]

# Input element to search
target = int(input("Enter element to search: "))

# Perform linear search
index = linear_search(numbers, target)

# Display result
if index != -1:
```

```
    print(f"Element {target} found at index {index}.")
else:
    print(f"Element {target} not found in the list.")
```

Output –

```
Enter element to search: 20
Element 20 found at index 1.
```

16. Write a Python Program for Binary Search Using a Recursive approach.

Program code:

```
def binary_search(arr, target, low, high):
    if low > high:
        return -1  # Element not found

    mid = (low + high) // 2

    if arr[mid] == target:
        return mid  # Element found
    elif arr[mid] > target:
        # Search in the left half
        return binary_search(arr, target, low, mid - 1)
    else:
        # Search in the right half
        return binary_search(arr, target, mid + 1, high)


# -------- MAIN PROGRAM --------

# Sorted list
numbers = [10, 20, 30, 40, 50, 60, 70]

# Input element to search
target = int(input("Enter element to search: "))

# Call recursive binary search
index = binary_search(numbers, target, 0, len(numbers) - 1)

# Display result
if index != -1:
    print(f"Element {target} found at index {index}.")
else:
    print(f"Element {target} not found in the list.")
```

Output –

```
Enter element to search: 60
Element 60 found at index 5.
```

17. Write a Python Program for Binary Search Using an Iterative approach.

Program code:

```
def binary_search_iterative(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid  # Element found
        elif arr[mid] < target:
            low = mid + 1  # Search in the right half
        else:
            high = mid - 1  # Search in the left half

    return -1  # Element not found


# -------- MAIN PROGRAM --------

# Sorted list
numbers = [10, 20, 30, 40, 50, 60, 70]

# Input element to search
target = int(input("Enter element to search: "))

# Perform iterative binary search
index = binary_search_iterative(numbers, target)

# Display result
if index != -1:
    print(f"Element {target} found at index {index}.")
else:
    print(f"Element {target} not found in the list.")
```

Output –

```
Enter element to search: 70
Element 70 found at index 6.
```

18. Write a Python Program for bubble sort.

Program code:

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n - 1):
        # Last i elements are already in place
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                # Swap if the element found is greater
                arr[j], arr[j + 1] = arr[j + 1], arr[j]


# -------- MAIN PROGRAM --------

numbers = [64, 34, 25, 12, 22, 11, 90]

print("Original list:", numbers)

# Perform bubble sort
bubble_sort(numbers)

print("Sorted list in ascending order:", numbers)
```

Output –

```
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list in ascending order: [11, 12, 22, 25, 34, 64, 90]
```

19. Write a Python Program for the implementation of merge sort.

Program code:

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

    # Merge elements in sorted order
    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

    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result


# Merge sort function
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)


# -------- MAIN PROGRAM --------

numbers = [38, 27, 43, 3, 9, 82, 10]

print("Original list:", numbers)

# Perform merge sort
sorted_list = merge_sort(numbers)

print("Sorted list in ascending order:", sorted_list)
```

Output –

```
Original list: [38, 27, 43, 3, 9, 82, 10]
Sorted list in ascending order: [3, 9, 10, 27, 38, 43, 82]
```

20. Write a Python Program for implementation of quick sort.

Program code:

```
def partition(arr, low, high):
    pivot = arr[high]  # Choosing the last element as pivot
    i = low - 1        # Index of smaller element

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]  # Swap

    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


# Quick sort function
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        # Recursively sort elements before and after partition
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)


# -------- MAIN PROGRAM --------

numbers = [10, 7, 8, 9, 1, 5]

print("Original list:", numbers)

# Perform quick sort
quick_sort(numbers, 0, len(numbers) - 1)

print("Sorted list in ascending order:", numbers)
```

Output –

```
Original list: [10, 7, 8, 9, 1, 5]
Sorted list in ascending order: [1, 5, 7, 8, 9, 10]
```