1. #Python Programs for Bubble Sort and Insertion Sort

Given the following elements 10, 7, 26, 43, 3, 13, 11

def bubble\_sort(arr):

    n = len(arr)

    for i in range(n):

        for j in range(0, n-i-1):

            if arr[j] > arr[j+1]:

                arr[j], arr[j+1] = arr[j+1], arr[j]

    return arr

# Test the bubble sort function

elements = [10, 7, 26, 43, 3, 13, 11]

sorted\_elements\_bubble = bubble\_sort(elements.copy())

print("Bubble Sort:", sorted\_elements\_bubble)

Bubble Sort: [3, 7, 10, 11, 13, 26, 43]

Insertion Sort

#Insertion sort builds the final sorted array one item at a time.

#It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.#

def insertion\_sort(arr):

    for i in range(1, len(arr)):

        key = arr[i]

        j = i - 1

        while j >= 0 and key < arr[j]:

            arr[j + 1] = arr[j]

            j -= 1

        arr[j + 1] = key

    return arr

 #Test the insertion sort function

sorted\_elements\_insertion = insertion\_sort(elements.copy())

print("Insertion Sort:", sorted\_elements\_insertion)

Insertion Sort: [3, 7, 10, 11, 13, 26, 43]

2. #Heap Sort with Iterations

#Heap sort involves building a heap from the input data and then repeatedly extracting the maximum element from the heap and reconstructing the heap.

#Here, I will show each step of the heap construction and sort process.

def heapify(arr, n, i):

    largest = i  # Initialize largest as root

    left = 2 \* i + 1  # left = 2\*i + 1

    right = 2 \* i + 2  # 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 heap\_sort(arr):

    n = len(arr)

    # Build a maxheap

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

        heapify(arr, n, i)

        print("Build Max Heap:", arr)

    # One by one extract elements

    for i in range(n-1, 0, -1):

        arr[i], arr[0] = arr[0], arr[i]  # swap

 print("Swap and Extract Max:", arr)

        heapify(arr, i, 0)

        print("Heapify:", arr)

    return arr

# Test the heap sort function

sorted\_elements\_heap = heap\_sort(elements.copy())

print("Heap Sort:", sorted\_elements\_heap)

Build Max Heap: [10, 7, 26, 43, 3, 13, 11]

Build Max Heap: [10, 43, 26, 7, 3, 13, 11]

Build Max Heap: [43, 10, 26, 7, 3, 13, 11]

Swap and Extract Max: [11, 10, 26, 7, 3, 13, 43]

Heapify: [26, 10, 13, 7, 3, 11, 43]

Swap and Extract Max: [11, 10, 13, 7, 3, 26, 43]

Heapify: [13, 10, 11, 7, 3, 26, 43]

Swap and Extract Max: [3, 10, 11, 7, 13, 26, 43]

Heapify: [11, 10, 3, 7, 13, 26, 43]

Swap and Extract Max: [7, 10, 3, 11, 13, 26, 43]

Heapify: [10, 7, 3, 11, 13, 26, 43]

Swap and Extract Max: [3, 7, 10, 11, 13, 26, 43]

Heapify: [7, 3, 10, 11, 13, 26, 43]

Swap and Extract Max: [3, 7, 10, 11, 13, 26, 43]

Heapify: [3, 7, 10, 11, 13, 26, 43]

Heap Sort: [3, 7, 10, 11, 13, 26, 43]

# Binary Search Program

def binary\_search(arr, x):

    low = 0

    high = len(arr) - 1

    mid = 0

    while low <= high:

        mid = (high + low) // 2

        if arr[mid] < x:

            low = mid + 1

        elif arr[mid] > x:

            high = mid - 1

        else:

            return mid

    return -1

# Test the binary search function

sorted\_elements = sorted(elements)  # Binary search requires a sorted array

element\_to\_search = 13

index = binary\_search(sorted\_elements, element\_to\_search)

print(f"Binary Search: Element {element\_to\_search} is at index {index} in sorted list {sorted\_elements}")

Binary Search: Element 13 is at index 4 in sorted list [3, 7, 10, 11, 13, 26, 43]

Binary Search: Element 13 is at index 4 in sorted list [3, 7, 10, 11, 13, 26, 43]

4.# Explanation of Linear Search Algorithm

Linear search is the simplest search algorithm. It checks every element of the list until it finds a match or reaches the end of the list.

Algorithm:

Start at the first element of the list.

Compare the current element with the target element.

If the current element is the target element, return its index.

If the current element is not the target element, move to the next element.

Repeat steps 2-4 until the target element is found or the end of the list is reached.

If the end of the list is reached and the target element has not been found, return -1 indicating the element is not in the list.

Time Complexity: O(n) in the worst case, where n is the number of elements in the list.