19CSE302 – Design and Analysis of Algorithms
Lab Assignment 2

Name: Pradeep Kumar Gupta
Reg no: -BL.EN.U4CSE21163

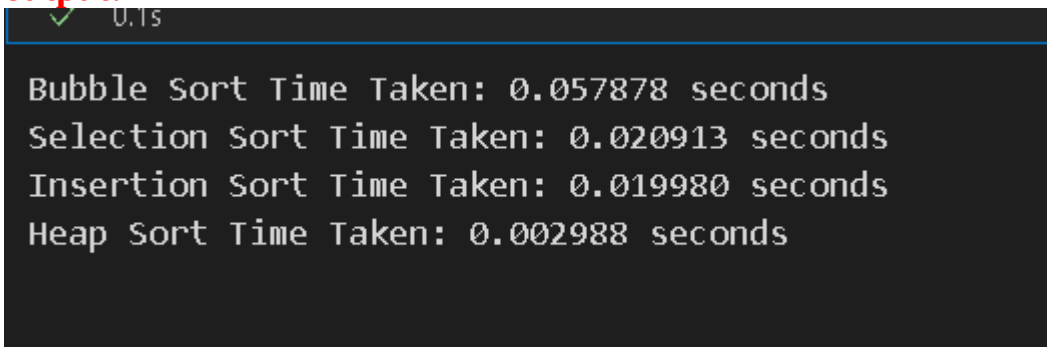
1.Generate 1000 integer random numbers between 1 and 10000. Compare the sorting algorithms learnt in the class using the same set of numbers generated. Print the time taken for them to complete the process.

```
import random
import time

# Generate 1000 random integers between 1 and 10,000
random_numbers = [random.randint(1, 10000) for _ in range(1000)]

# Define sorting algorithms

# Bubble Sort
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Selection Sort
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Insertion 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

# Heap Sort
def heap_sort(arr):
    def heapify(arr, n, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

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

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

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

    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)

# Measure the time taken for each sorting algorithm

# Bubble Sort
bubble_start_time = time.time()
bubble_sort(random_numbers.copy())
bubble_end_time = time.time()
bubble_time_taken = bubble_end_time - bubble_start_time

# Selection Sort
selection_start_time = time.time()
selection_sort(random_numbers.copy())
selection_end_time = time.time()
selection_time_taken = selection_end_time - selection_start_time

# Insertion Sort
insertion_start_time = time.time()
insertion_sort(random_numbers.copy())
insertion_end_time = time.time()
```

```
insertion_time_taken = insertion_end_time - insertion_start_time

# Heap Sort
heap_start_time = time.time()
heap_sort(random_numbers.copy())
heap_end_time = time.time()
heap_time_taken = heap_end_time - heap_start_time

# Print the time taken for each sorting algorithm
print(f"Bubble Sort Time Taken: {bubble_time_taken:.6f} seconds")
print(f"Selection Sort Time Taken: {selection_time_taken:.6f} seconds")
print(f"Insertion Sort Time Taken: {insertion_time_taken:.6f} seconds")
print(f"Heap Sort Time Taken: {heap_time_taken:.6f} seconds")
```

output:-



2.

Given ‘m’ sorted lists/ arrays, each containing ‘n’ elements, print them efficiently in sorted order

```
import heapq

# Define the input sorted lists
sorted_lists = [
    [10, 20, 30, 40],
    [15, 25, 35],
    [27, 29, 37, 48, 93],
    [32, 33]
]

# Initialize a result list to store the sorted elements
result = []

# Create a min-heap and add the first element from each sorted list
min_heap = []

for i, lst in enumerate(sorted_lists):
```

```
    if len(lst) > 0:
        heapq.heappush(min_heap, (lst[0], i, 0))

# Merge the sorted lists using the min-heap
while min_heap:
    val, lst_idx, ele_idx = heapq.heappop(min_heap)
    result.append(val)

    # Check if there is another element in the same list
    if ele_idx + 1 < len(sorted_lists[lst_idx]):
        heapq.heappush(min_heap, (sorted_lists[lst_idx][ele_idx + 1], lst_idx,
ele_idx + 1))

# Print the sorted result
print(result)
```

output:-

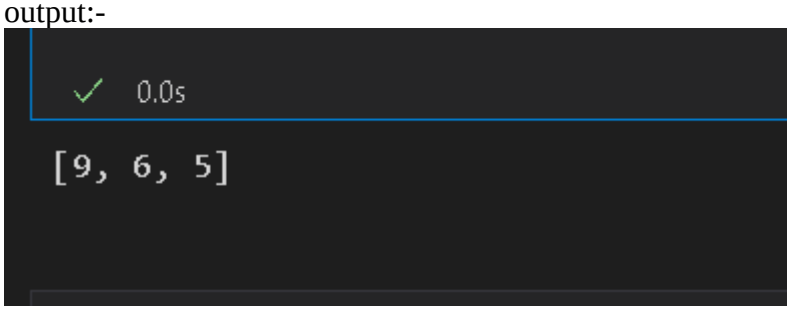
```
✓ 0.0s
[10, 15, 20, 25, 27, 29, 30, 32, 33, 35, 37, 40, 48, 93]
```

3. Given an array of size N, find the K largest elements in the array where K<<N

```
import heapq

def find_k_largest_elements(arr, k):
    # Create a min-heap with a size limit of K
    min_heap = []

    # Add the first K elements to the min-heap
    for i in range(k):
        heapq.heappush(min_heap, arr[i])

    # For the remaining elements, compare with the smallest element in the
min-heap
    # If larger, replace the smallest element with the current element
    for i in range(k, len(arr)):
        if arr[i] > min_heap[0]:
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, arr[i])

    # The min-heap now contains the K largest elements
    return sorted(min_heap, reverse=True)

# Example usage:
arr = [3, 1, 4, 1, 5, 9, 2, 6]
k = 3
```

```
result = find_k_largest_elements(arr, k)
print(result)  # Output: [9, 6, 5]
```

output:-

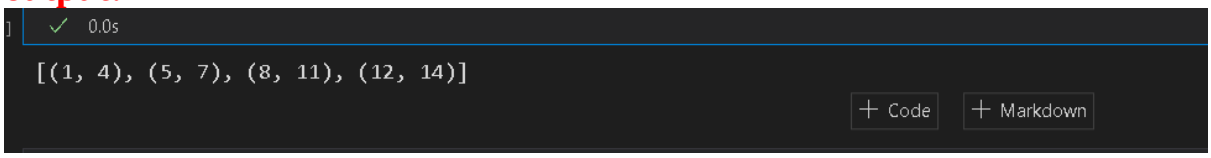


4. Given a set of activities, along with the starting and finishing time of each activity, find the maximum number of activities performed by a single person assuming that a person can only work on a single activity at a time.

```
def activity_selection(activities):
    # Sort activities by finish time
    activities.sort(key=lambda x: x[1])

    selected_activities = []
    n = len(activities)

    # The first activity always gets selected
    selected_activities.append(activities[0])

    # Initialize the index of the most recent selected activity
    prev_activity_index = 0

    # Consider the rest of the activities
    for current_activity_index in range(1, n):
        # If this activity has a start time greater than or equal to the
finish time
        # of the previously selected activity, add it to the selected
activities
        if activities[current_activity_index][0] >=
activities[prev_activity_index][1]:
            selected_activities.append(activities[current_activity_index])
            prev_activity_index = current_activity_index

    return selected_activities

# Input: List of activities as tuples (start_time, finish_time)
activities =
[(1,4),(3,5),(0,6),(5,7),(3,8),(5,9),(6,10),(8,11),(8,12),(2,13),(12,14)]

# Find the maximum number of activities performed
selected_activities = activity_selection(activities)
```

```
# Print the selected activities
print(selected_activities)
```

output:-



5. Given a set of intervals, print all non-overlapping intervals after merging the overlapping intervals

```
def merge_intervals(intervals):
    if not intervals:
        return []

    # Sort the intervals based on their start times
    intervals.sort(key=lambda x: x[0])

    merged_intervals = [intervals[0]]

    for i in range(1, len(intervals)):
        current_interval = intervals[i]
        previous_interval = merged_intervals[-1]

        if current_interval[0] <= previous_interval[1]:
            # Overlapping intervals: merge them
            merged_intervals[-1] = (previous_interval[0],
max(previous_interval[1], current_interval[1]))
        else:
            # Non-overlapping interval: add it to the result
            merged_intervals.append(current_interval)

    return merged_intervals

# Input: List of intervals as tuples (start, end)
intervals = [(1, 4), (2, 5), (7, 8), (6, 9)]

# Merge the overlapping intervals
merged_intervals = merge_intervals(intervals)

# Print the merged intervals
print(merged_intervals)
```

output:-


0.0s
[(1, 5), (6, 9)]