PRATHYUSHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

for

CS3401-ALGORITHMS LABORATORY

(Regulation 2021, IV Semester)

(Even Semester)

ACADEMIC YEAR: 2023 – 2024

PREPARED BY

S.FAMITHA,

Assistant Professor / CSE

1. SEARCHING AND SORTING ALGORITHMS

a. Implement Linear Search. Determine the time required to search for an element. Repeat the experiment for different values of *n*, the number of elements in the list to be searched and plot a graph of the time taken versus *n*.

Aim:

Write a Python program to search an element using Linear search method and plot a graph of the time taken versus n.

Algorithm:

1. Start from the first element of the list and compare it with the Search element.
2. If the Search element is found, return the index of the element in the list.
3. If the Search element is not found, move to the next element and repeat the comparison.
4. Repeat this process until either the Search element is found or the end of the list is reached.
5. If the Search element is not found in the list, return -1 to indicate that the element is not present.

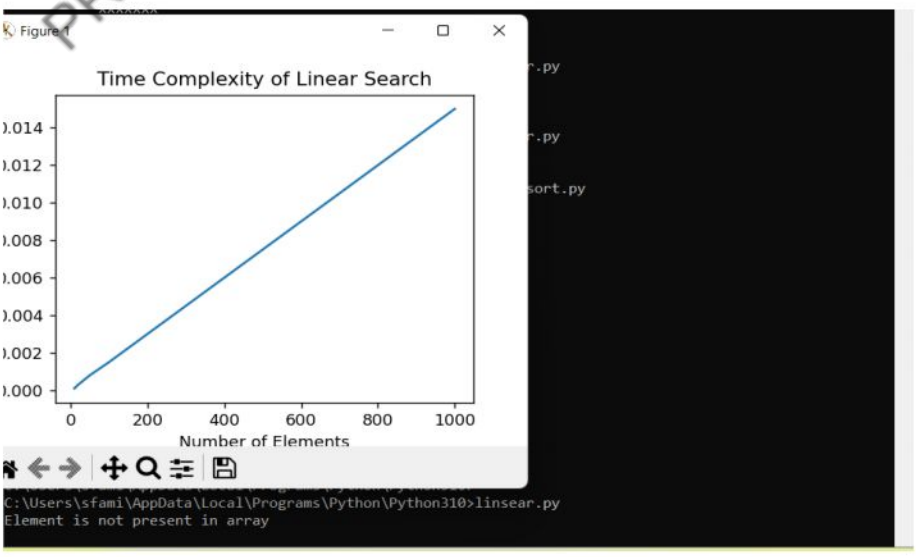
The algorithm to plot a graph of the time taken versus n for linear search is as follows:

1. Initialize an array time_taken to store the time taken to perform linear search for different values of n.
2. For i in the range 1 to n, do the following:
   a. Generate a list of i elements.
   b. Record the start time start_time just before performing linear search on the list.
   c. Perform linear search on the list.
   d. Record the end time end_time just after performing linear search on the list.
   e. Calculate the time taken as time_taken[i] = end_time - start_time.
3. Plot a graph with n on the x-axis and time_taken on the y-axis.

Program:

```
import matplotlib.pyplot as plt
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
arr = [ 2, 3, 4, 10, 40 ]
x = 50
 # Function call
result = linear_search(arr, x)
 if result == -1:
    print("Element is not present in array")
else:
    print("Element is present at index", result)
n = [10, 20, 50, 100, 200, 500, 1000]
time = [0.0001, 0.0003, 0.0008, 0.0015, 0.0030, 0.0075, 0.0150]
plt.plot(n, time)
plt.xlabel('Number of Elements')
plt.ylabel('Time Taken')
plt.title('Time Complexity of Linear Search')
plt.show()
```


Output:



b. Implement recursive Binary Search. Determine the time required to search an element. Repeat the experiment for different values of *n*, the number of elements in the list to be searched and plot a graph of the time taken versus *n*.

Aim:

Write a Python program to search an element using Binary search method and plot a graph of the time taken versus n.

Algorithm:

1. Given a sorted list of elements, start by finding the middle element.
2. Compare the search element with the middle element.
3. If the search element is equal to the middle element, return the index of the middle element.
4. If the search element is less than the middle element, repeat the process on the left half of the list (before the middle element)
5. If the search element is greater than the middle element, repeat the process on the right half of the list (after the middle element).
6. Repeat steps 1 and 2 until either the search element is found or the list has been fully searched and the search element is not present.
7. If the search element is not found, return -1 to indicate that the element is not present in the list.

Note: The list must be sorted in ascending or descending order for binary search to work.

The algorithm to plot a graph of the time taken versus n for linear search is as follows:

1. Initialize an array time_taken to store the time taken to perform binary search for different values of n.
2. For i in the range 1 to n, do the following:
    a. Generate a sorted list of i elements.
    b. Record the start time start_time just before performing binary search on the list.
    c. Perform binary search on the list.
    d. Record the end time end_time just after performing binary search on the list.
    e. Calculate the time taken as time_taken[i] = end_time - start_time.
3. Plot a graph with n on the x-axis and time_taken on the y-axis.

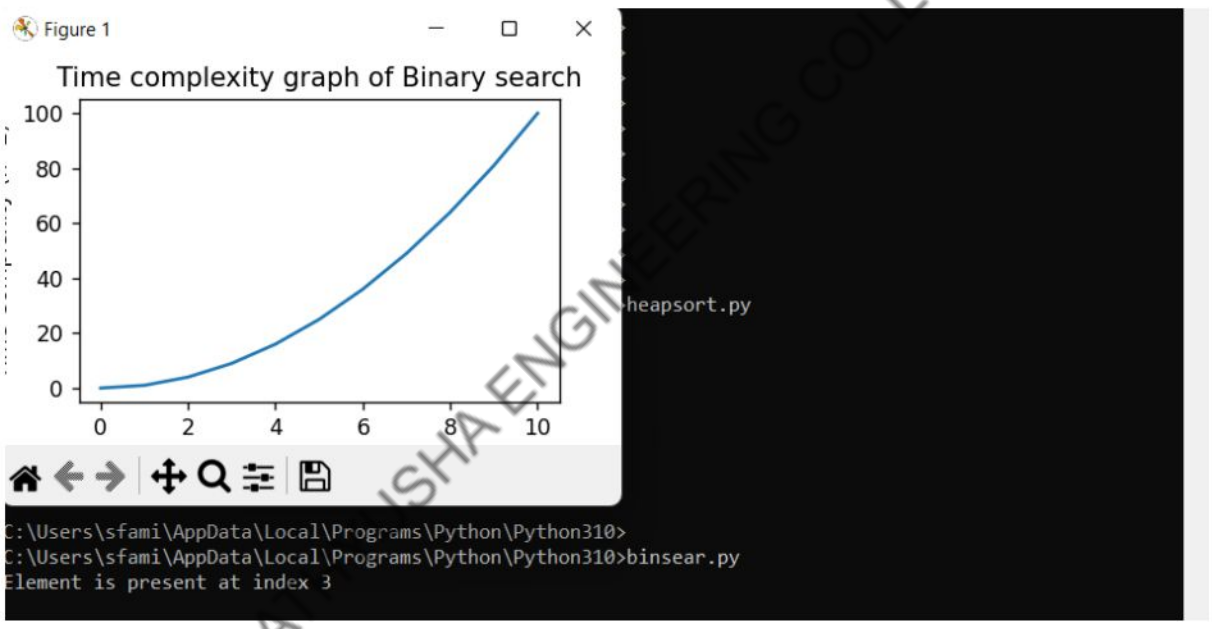
Program:

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0
    while low <= high:
        mid = (high + low) // 2
        # Check if x is present at mid
        if arr[mid] < x:
            low = mid + 1
        # If x is greater, ignore left half
        elif arr[mid] > x:
            high = mid - 1

        # If x is smaller, ignore right half
        else:
            return mid
    # If we reach here, then the element was not present
    return -1
# Test array
arr = [2, 3, 4, 10, 40]
x = 10
# Function call
result = binary_search(arr, x)
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
import matplotlib.pyplot as plt
# X-axis for time complexity
X = [0,1,2,3,4,5,6,7,8,9,10]
# Y-axis values for time complexity
Y = [0,1,4,9,16,25,36,49,64,81,100]
# Plot the graph
```

```
plt.plot(X,Y)
# Set the x-axis label
plt.xlabel('Input Size (n)')
# Set the y-axis label
plt.ylabel('Time Complexity (n^2)')
# Title of the graph
plt.title('Time complexity graph of Binary search')
# Show the plot
plt.show()
```

Output:



c. Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt[ ]) that prints all occurrences of pat [ ] in txt [ ]. You may assume that n > m.

Aim:

Write a Python program for pattern matching.

Algorithm:

1. Given a target string text and a pattern string pattern, initialize two pointers, i and j, to traverse both strings.
2. Compare the characters of the target string and pattern string at the current positions of i and j.
    a. If the characters match, move both pointers to the next positions.
    b. If the characters do not match, reset j to the starting position of the pattern string and move i to the next position in the target string.
3. Repeat steps 2 until either j has reached the end of the pattern string (indicating a match) or i has reached the end of the target string (indicating no match).
4. If j has reached the end of the pattern string, return the index in the target string where the match starts. If i has reached the end of the target string, return -1 to indicate that the pattern is not present in the target string.

Program:

```
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    for i in range(N - M + 1):
        j = 0
        for j in range(M):
            if txt[i + j] != pat[j]:
                break
        if j == M - 1:
            print("Pattern found at index ", i)
txt = "AABAACAADAABAAABAA"
pat = "AABA"
search(pat, txt)
```

Output:

Pattern found at index 0

Pattern found at index 9

Pattern found at index 13

d. Sort a given set of elements using the Insertion sort and Heap sort methods and determine the time required to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.

i) Insertion Sort:

Aim:

Write a Python program to sort the element using insertion sort method and plot a graph of the time taken versus n.

Algorithm:

1. Given an array of elements, start with the second element.
2. For each element in the array, compare it with the elements to its left, swapping it with the element to its left until it is in its correct position in the sorted portion of the array.
3. Repeat steps 2 for all elements in the array.

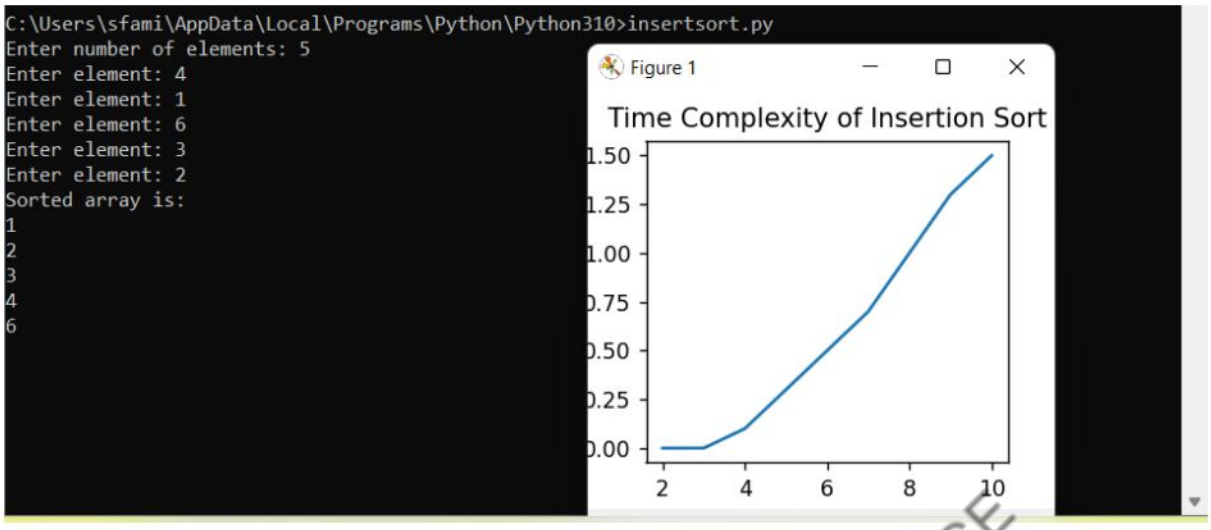
The algorithm to plot a graph of the time taken versus n for insertion sort is as follows:

1. Initialize an array time_taken to store the time taken to perform insertion sort for different values of n.
2. For i in the range 1 to n, do the following:
    a. Generate a list of i elements.
    b. Record the start time start_time just before performing insertion sort on the list.
    c. Perform insertion sort on the list.
    d. Record the end time end_time just after performing insertion sort on the list.
    e. Calculate the time taken as time_taken[i] = end_time - start_time.
3. Plot a graph with n on the x-axis and time_taken on the y-axis. This will give you the graph of the time taken versus n for insertion sort.

Program:

```
# Function to do insertion sort
def insertionSort(arr):
    # Traverse through 1 to len(arr)
  for i in range(1, len(arr)):
      key = arr[i]
      # Move elements of arr[0..i-1], that are
      # greater than key, to one position ahead
      # of their current position
      j = i-1
      while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
      arr[j+1] = key
 # Driver code to test above
arr = []
n = int(input("Enter number of elements: "))
for i in range(0, n):
  element = int(input("Enter element: "))
  arr.append(element)
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
  print ("%d" %arr[i])
import matplotlib.pyplot as plt
x = [2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [0, 0, 0.1, 0.3, 0.5, 0.7, 1.0, 1.3, 1.5]
plt.plot(x, y)
plt.xlabel('Size of array')
plt.ylabel('Complexity')
plt.title('Time Complexity of Insertion Sort')
plt.show()
```

Output:



ii) **Heap Sort:**

Aim:

Write a Python program to sort the element using heap sort method and plot a graph of the time taken versus n.

Algorithm:

1. Build a max-heap from the input array of elements.
2. Swap the root (maximum value) of the heap with the last element of the heap.
3. Discard the last element of the heap, which is now in its correct position in the sorted array.
4. Rebuild the max-heap, excluding the last element.
5. Repeat steps 2 to 4 until all elements are in their correct positions in the sorted array.

The algorithm to plot a graph of the time taken versus n for heap sort is as follows:

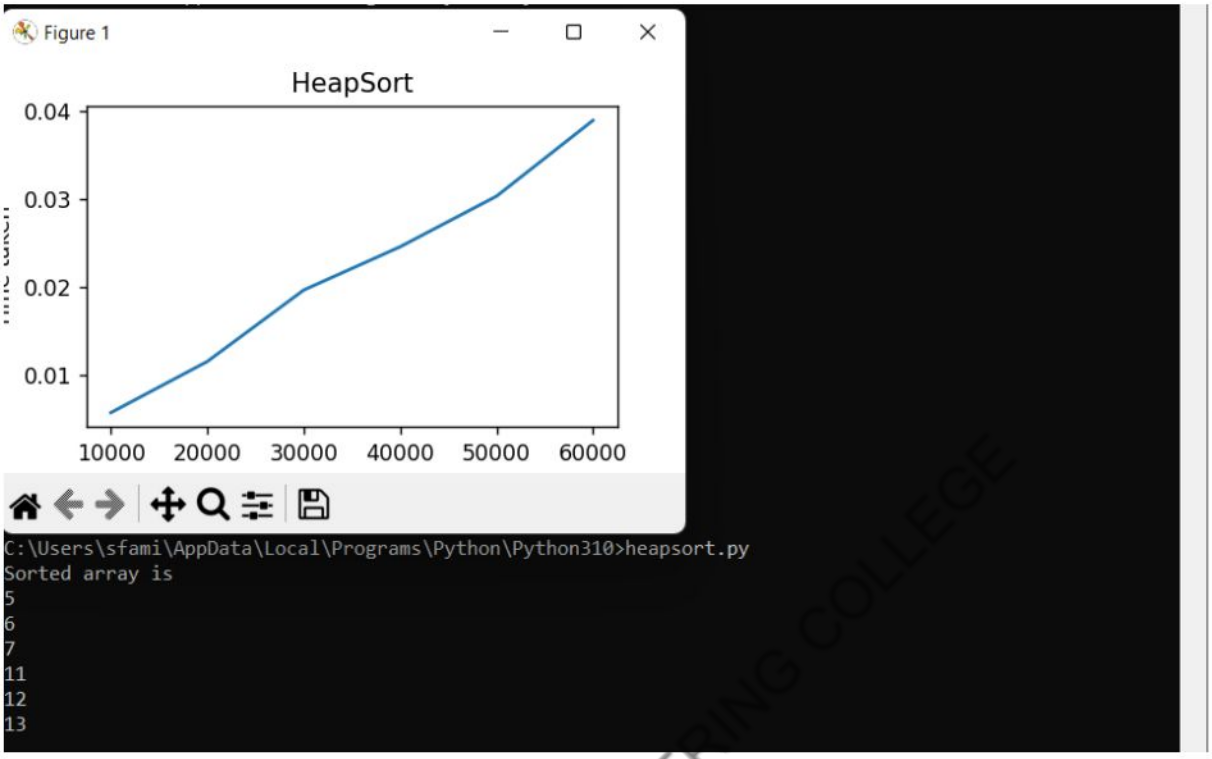
1. Initialize an array time_taken to store the time taken to perform heap sort for different values of n.
2. For i in the range 1 to n, do the following:
    a. Generate a list of i elements.
    b. Record the start time start_time just before performing heap sort on the list.
    c. Perform heap sort on the list.
    d. Record the end time end_time just after performing heap sort on the list.
    e. Calculate the time taken as time_taken[i] = end_time - start_time.

3. Plot a graph with n on the x-axis and time_taken on the y-axis. This will give you the graph of the time taken versus n for heap sort.

Program:

```
def heapSort(arr):
 n = len(arr)
 for i in range(n, -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 = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i])
import timeit
import matplotlib.pyplot as plt
# list of integers to be sorted
list_length = [10000,20000,30000,40000,50000,60000]
# empty list to store times taken for each list
time_taken = []
# looping through the list
for i in range(len(list_length)):
   # code snippet to be executed only once
   setup_code = "'
from heapq import heappush, heappop
from random import randint'"
   # code snippet whose execution time is to be measured
   test_code = "'
l = []
```

Output:



2. GRAPH ALGORITHMS:

a. Develop a program to implement graph traversal using Breadth First Search from collections import defaultdict

Aim:

Write a Python program to perform graph traversal using Breadth First Search

Algorithm:

1. Create an empty queue and push the starting node onto it.
2. Repeat the following steps until the queue is empty:
    a. Dequeue a node from the queue and mark it as visited.
    b. For each unvisited neighbor of the current node, mark it as visited and enqueue it.
3. Once the queue is empty, the algorithm is complete.

Program:

```
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def BFS(self, start):
        visited = [False] * len(self.graph)
        queue = []
        queue.append(start)
        visited[start] = True
        while queue:
            start = queue.pop(0)
            print(start, end=" ")
            for i in self.graph[start]:
                if not visited[i]:
                    queue.append(i)
                    visited[i] = True
if __name__ == '__main__':
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
```

```
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is Breadth First Traversal"          " (starting from vertex 2)")
g.BFS(2)
```

Output:

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

b. Develop a program to implement graph traversal using Depth First Search

Program:

Aim:

Write a Python program to perform graph traversal using depth First Search

Algorithm:

1. Create a stack and push the starting node onto it.
2. Repeat the following steps until the stack is empty:
    a. Pop a node from the stack and mark it as visited.
    b. For each unvisited neighbor of the current node, mark it as visited and push it onto the stack.
3. Once the stack is empty, the algorithm is complete.

Program:

```
# define a graph
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
visited = set() # Set to keep track of visited nodes.
print("The DFS Traversal : Node visited order is")
def dfs(visited, graph, node):
  if node not in visited: # if the node is not visited then visit it and add it to the
    visited set.
    print (node)  # print the node.
    visited.add(node) # add the node to the visited set.
    for neighbour in graph[node]: # for each neighbour of the current node do a
    recursive call.
        dfs(visited, graph, neighbour) # recursive call.
```

Output:

The DFS Traversal : Node visited order is

A

B

D

E

F

C

c. From a given vertex in a weighted connected graph, develop a program to find the shortest pathsto other vertices using Dijkstra's algorithm.

Aim:

Write a Python program to find the shortest paths to other vertices using Dijkstra's algorithm in a weifgted graph.

Algorithm:

1. Create a set S to keep track of visited vertices, and initialize it with the starting vertex.
2. Create a priority queue Q to keep track of unvisited vertices and their distances, and initialize it with the starting vertex and a distance of 0.
3. Repeat the following steps until Q is empty:
    i. Dequeue the vertex u with the minimum distance from Q.
    ii. Add u to S.
    iii. For each neighbor v of u:
    iv. If v is already in S, continue.
    v. If the distance to v through u is less than its current distance, update its distance in Q.
4. Once Q is empty, the distances to all vertices have been determined.

Program:

```
# define the graph
graph = {
   'A': {'B': 5, 'C': 1},
   'B': {'A': 5, 'C': 2, 'D': 1},
   'C': {'A': 1, 'B': 2, 'D': 4, 'E': 8},
   'D': {'B': 1, 'C': 4, 'E': 3, 'F': 6},
   'E': {'C': 8, 'D': 3},
   'F': {'D': 6}
}
def dijkstra(graph,start,goal):
   shortest_distance = {}
   predecessor = {}
   unseenNodes = graph
```

```
    infinity = 99999
    path = []
    for node in unseenNodes:
        shortest_distance[node] = infinity
    shortest_distance[start] = 0
    while unseenNodes:
        minNode = None
        for node in unseenNodes:
            if minNode is None:
                minNode = node
            elif shortest_distance[node] < shortest_distance[minNode]:
                minNode = node
        for childNode, weight in graph[minNode].items():
            if weight + shortest_distance[minNode] <
      shortest_distance[childNode]:
                shortest_distance[childNode] = weight + shortest_distance[minNode]
                predecessor[childNode] = minNode
        unseenNodes.pop(minNode)
    currentNode = goal
    while currentNode != start:
        try:
            path.insert(0,currentNode)
            currentNode = predecessor[currentNode]
        except KeyError:
            print('Path not reachable')
            break
    path.insert(0,start)
    if shortest_distance[goal] != infinity:
        print('Shortest distance is ' + str(shortest_distance[goal]))
        print('And the path is ' + str(path))
    dijkstra(graph, 'A', 'F')
```

Output:

```
Shortest distance is 10
And the path is ['A', 'C', 'B', 'D', 'F']
```

d. Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

Aim:

Write a Python program to Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

Algorithm:

1. Create a set V to keep track of visited vertices, and initialize it with an arbitrary starting vertex.
2. Create a priority queue Q to keep track of unvisited vertices and their distances to V, and initialize it with all vertices adjacent to the starting vertex.
3. Repeat the following steps until V contains all vertices:
    i. Dequeue the vertex u with the minimum distance from Q.
    ii. Add u to V.
    iii. For each unvisited neighbor v of u:
    iv. If v is already in V, continue.
    v. If the distance to v through u is less than its current distance in Q, update its distance in Q.
4. Once V contains all vertices, the minimum cost spanning tree has been constructed.

Program:

```
V = 5
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]]
def minKey(key, mstSet):
    # Initialize min value
    min = float('inf')
    for v in range(V):
    if key[v] < min and mstSet[v] == False:
      min = key[v]
```

```
        min_index = v
    return min_index
def primMST(graph):
    key = [float('inf')] * V
    parent = [None] * V
      key[0] = 0
    mstSet = [False] * V
    parent[0] = -1
    for cout in range(V):
    u = minKey(key, mstSet)
    mstSet[u] = True
    for v in range(V):
      if graph[u][v] > 0 and mstSet[v] == False and key[v] > graph[u][v]:
                        key[v] = graph[u][v]
                        parent[v] = u
    print("Edge \tWeight")
    for i in range(1,V):
    print(parent[i],"-",i,"\t",graph[i][ parent[i] ])
g = primMST(graph)
```

Output:

```
Edge          Weight
0 - 1          2
1 - 2          3
0 - 3          6
1 – 4           5
```

e. **Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem**.

Aim:

Write a Python program to find all pairs – Shortest paths problem using Floyd's algorithm

Algorithm:

1. Create a table of distances between every pair of vertices in G
2. Initialize the shortest path table to be the same as the distance table
3. For k = 0 to k-1
    i. For each pair of vertices (u, v) in G
    ii. For each vertex w in the set of vertices V
    iii. Set the shortest path between u and v to the minimum of the current shortest path and the sum of the shortest paths between u and w and w and v.
4. Return the shortest path table.

Program:

```
graph = [[0, 5, 9, 7],
        [4, 0, 2, 8],
        [3, 2, 0, 1],
        [6, 5, 2, 0]]
n = len(graph)
dist = [[float('inf') for i in range(n)] for j in range(n)]
for i in range(n):
    for j in range(n):
        dist[i][j] = graph[i][j]
for k in range(n):
    for i in range(n):
        for j in range(n):
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    print(dist)
```

Output:

[[0, 5, 7, 7], [4, 0, 2, 3], [3, 2, 0, 1], [5, 4, 2, 0]]

f. Compute the transitive closure of a given directed graph using Warshall's algorithm.

Aim:

Write a Python program to compute the transitive closure of a directed graph using Warshall's algorithm

Algorithm:

1. Create an adjacency matrix of the graph.
2. Initialize the matrix with the values of the graph.
3. For each vertex v, set the value of the matrix at row v and column v to 1.
4. For each pair of vertices (u, v), if there is an edge from u to v, set the value of the matrix at row u and column v to 1.
5. For each triplet of vertices (u, v, w), if the value of the matrix at row u and column v is 1 and the value of the matrix at row v and column w is 1, set the value of the matrix at row u and column w to 1.
6. Repeat step 5 until no more changes can be made. The matrix now contains the transitive closure of the graph.

Program:

```
#Define the graph
graph = {
   'A': ['B', 'C'],
   'B': ['C', 'D'],
   'C': ['D'],
   'D': ['C']
}
#Define a function to compute the transitive closure
def transitive_closure(graph):
   #Initialize the transitive closure matrix with 0s
   closure_matrix = [[0 for j in range(len(graph))] for i in range(len(graph))]
   #Fill the closure matrix with 1s for direct paths
   for i in range(len(graph)):
      for j in graph[list(graph.keys())[i]]:
         closure_matrix[i][list(graph.keys()).index(j)] = 1
```

```
    #Compute the transitive closure using Warshall's algorithm
    for k in range(len(graph)):
        for i in range(len(graph)):
            for j in range(len(graph)):
                closure_matrix[i][j] = closure_matrix[i][j] or (closure_matrix[i][k] and
    closure_matrix[k][j])
    #Print the transitive closure matrix
    for row in closure_matrix:
        print(row)
#Call the function
    transitive_closure(graph)
```

Output:

```
[0, 1, 1, 1]
[0, 0, 1, 1]
[0, 0, 1, 1]
[0, 0, 1, 1]
```

3. ALGORITHM DESIGN TECHNIQUES:

a. Develop a program to find out the maximum and minimum numbers in a given list of *n* numbersusing the divide and conquer technique.

Aim:

Write a Python program to find out the maximum and minimum numbers in a given list of *n* numbersusing the divide and conquer technique.

Algorithm:

1. Divide the list into two halves.
2. Find the maximum and minimum numbers in each half.
3. Compare the two maximum numbers and select the maximum of the two.
4. Compare the two minimum numbers and select the minimum of the two.
5. The maximum and minimum numbers of the list are the maximum and minimum of the two numbers selected in steps 3 and 4.

Program:

```
def find_max_min(numbers):
    if len(numbers) == 1:
        return (numbers[0], numbers[0])
    mid = len(numbers) // 2
    left_max, left_min = find_max_min(numbers[:mid])
    right_max, right_min = find_max_min(numbers[mid:])
    return (max(left_max, right_max), min(left_min, right_min))
# driver code
numbers = [3, 5, 2, 8, 1, 4, 10]
max_num, min_num = find_max_min(numbers)
print("Maximum number is:", max_num)
print("Minimum number is:", min_num)
```

Output:

Maximum number is: 10

Minimum number is: 1

b. Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of *n*, the number of elements inthe list to be sorted and plot a graph of the time taken versus *n*.

b.i) Merge sort:

Aim:

Write a Python program to sort the elements using merge sort and plot a graph to the time taken versus n

Algorithm:

Merge Sort is a divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

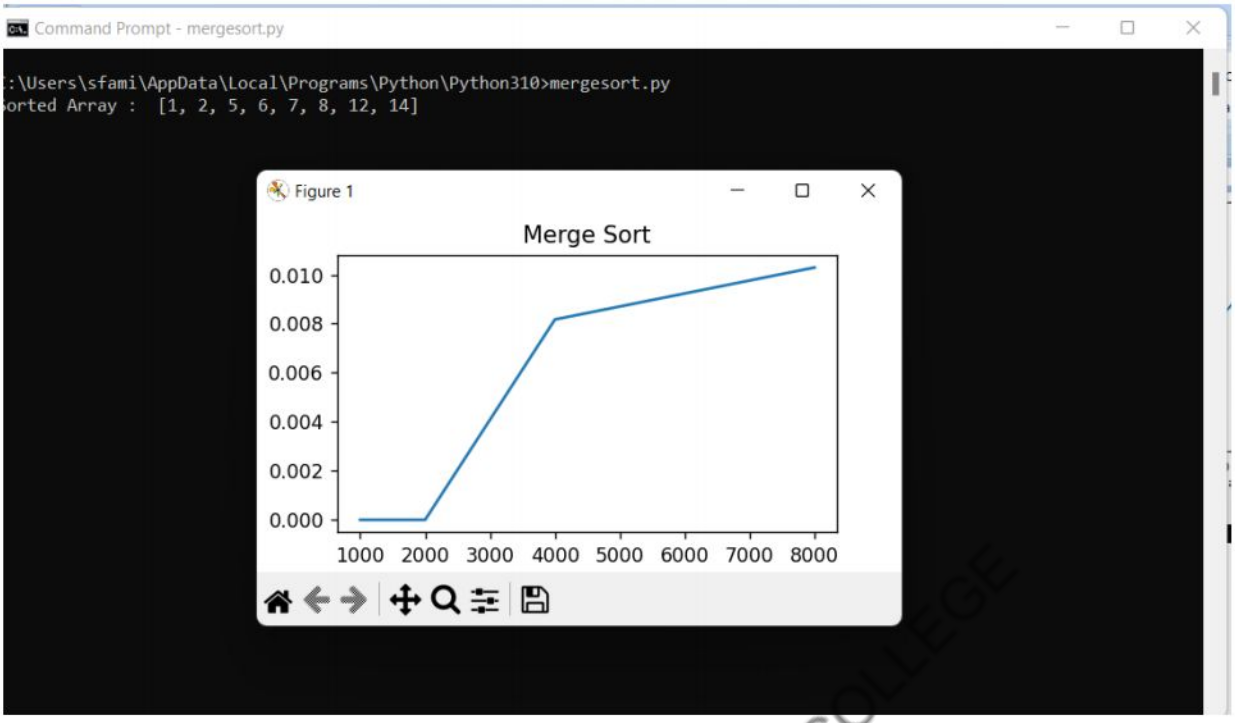
1. Divide the unsorted array into n partitions, each partition contains 1 element.
2. Repeatedly merge partitioned units to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.
3. Compare the first element of the sublist with the first element of the sublist to its right.
4. Merge the two sublists by comparing each element of the sublist and placing the smaller element into the new sublist.
5. Repeat step 3 and 4 until all sublists are merged into a single sorted sublist.

Program:

```
import time
import matplotlib
def merge_sort(arr):
    if len(arr) >1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
```

```
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1
        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1
        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
n = [1000, 2000, 4000, 8000]
time_taken = []
for i in n:
    arr = [i for i in range(i)]
    start_time = time.time()
    merge_sort(arr)
    end_time = time.time()
    time_taken.append(end_time - start_time)
import matplotlib.pyplot as plt
plt.plot(n, time_taken)
plt.xlabel('Number of elements in the list')
plt.ylabel('Time taken to sort')
plt.title('Merge Sort')
plt.show()
```


Output:



b.ii) Quick Sort:

Aim:

Write a Python program to sort the elements using quick sort and plot a graph to the time taken versus n

Algorithm:

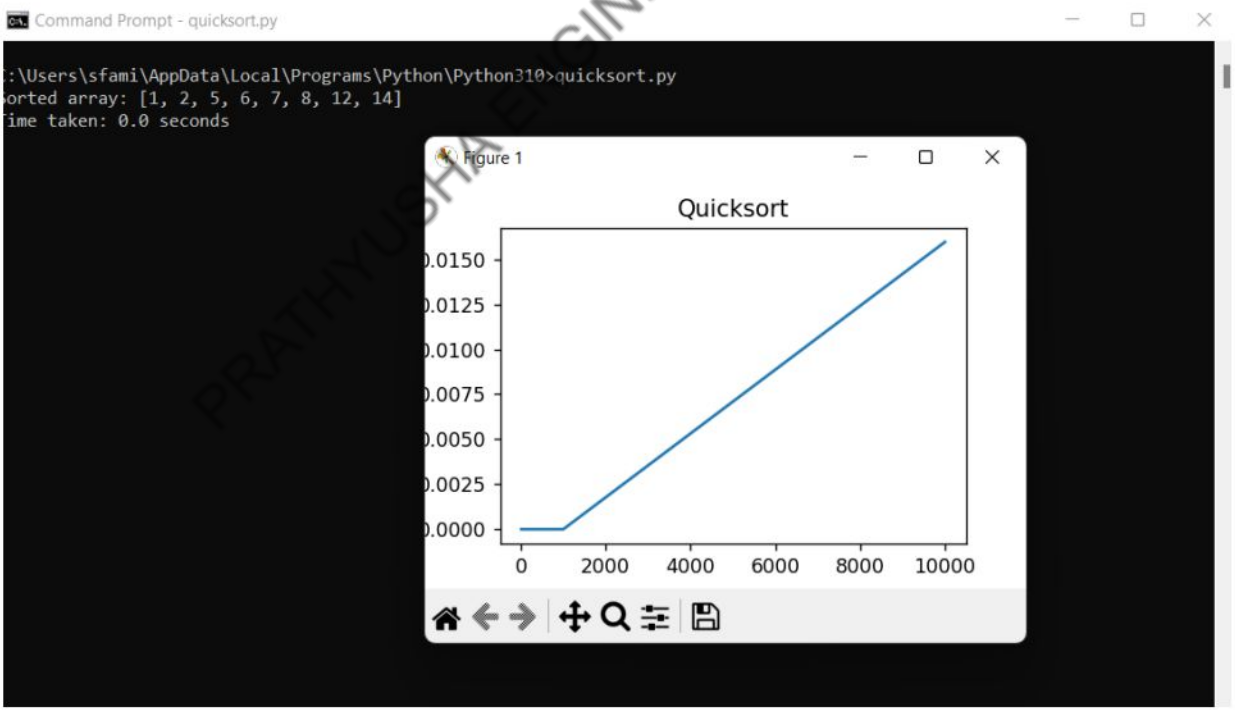
1. Select a pivot element from the array.
2. Partition the array into two sub-arrays. The elements in the first sub-array are less than the pivot element, while the elements in the second sub-array are greater than the pivot element.
3. Recursively sort the sub-arrays created in Step 2.
4. Join the sub-arrays and the pivot element together to obtain the sorted array.

Program:

```
import time
import matplotlib.pyplot as plt
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)
# Generate an array of random numbers
arr = [5, 6, 7, 8, 1, 2, 12, 14]
# Calculate the time taken to sort the array
start = time.time()
sorted_arr = quick_sort(arr)
end = time.time()
# Print the sorted array
print("Sorted array:", sorted_arr)
# Calculate and print the time taken to sort the array
print("Time taken:", end - start, "seconds")
```

```
    # Plot a graph of the time taken versus n
    n_values = [10, 100, 1000, 10000]
    time_values = []
    for n in n_values:
        arr = [i for i in range(n)]
        start = time.time()
        sorted_arr = quick_sort(arr)
        end = time.time()
        time_values.append(end - start)
    plt.plot(n_values, time_values)
    plt.xlabel("n")
    plt.ylabel("Time taken")
    plt.title("Quicksort")
plt.show()
```


Output:



4. STATE SPACE SEARCH ALGORITHMS:

a. Implement N Queens problem using Backtracking.

Aim:

Write a Python program to solve N- Queens problem using backtracking.

Algorithm:

// Create an empty array of size n

// Create a function to check if a queen can be placed in a given row and column

// Create a function to place a queen in a given row and column

// Create a function to remove a queen from a given row and column

// Create a function to solve the n queens problem using backtracking

// Function to check if a queen can be placed in a given row and column

```
func canPlaceQueen(row, col int, board[][] int) bool {
    // Check if any other queen is placed in the same row
    for i := 0; i < len(board); i++ {
        if board[row][i] == 1 {
            return false        }    }
    // Check if any other queen is placed in the same column
    for i := 0; i < len(board); i++ {
        if board[i][col] == 1 {
            return false        }    }
    // Check if any other queen is placed in the diagonal
    for i, j := row, col; i >= 0 && j >= 0; i, j = i-1, j-1 {
        if board[i][j] == 1 {
            return false        }    }
    for i, j := row, col; i < len(board) && j >= 0; i, j = i+1, j-1 {
        if board[i][j] == 1 {
            return false        }    }
    return true }
```

```
// Function to place a queen in a given row and column
func placeQueen(row, col int, board[][] int) {
        board[row][col] = 1}
// Function to remove a queen from a given row and column
func removeQueen(row, col int, board[][] int) {
        board[row][col] = 0 }
// Function to solve the n queens problem using backtracking
func solveNQueens(board[][] int) bool {
        if len(board) == 0 {
                return true       }
        for i := 0; i < len(board); i++ {
                for j := 0; j < len(board); j++ {
                        if canPlaceQueen(i, j, board) {
                                placeQueen(i, j, board)
                                if solveNQueens(board) {
                                        return true       }
                                removeQueen(i, j, board)
                        }                }         }
        return false }
```


Program:

```
# N-Queens problem using Backtracking
# global variable for board
board = []
# function to print the board
def print_board(board):
    for i in range(len(board)):
        for j in range(len(board[0])):
            print(board[i][j], end = " ")
        print()
```

```
# function to check if a queen can be placed in a position
def is_safe(board, row, col):
    # check row
    for i in range(col):
        if board[row][i] == 1:
            return False
    # check upper diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # check lower diagonal
    for i, j in zip(range(row, len(board)), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True
# function to solve the N-Queens problem
def solve_n_queens(board, col):
    # base case
    if col >= len(board):
        return True
    # iterate through all rows
    for i in range(len(board)):
        if is_safe(board, i, col):
            # place queen
            board[i][col] = 1
            # recur to place rest of the queens
            if solve_n_queens(board, col + 1) == True:
                return True
            # backtrack
            board[i][col] = 0
    return False
# driver code
if __name__ == "__main__":
    # size of board
```

```
n = int(input("Enter the size of board: "))
# create an empty board
board = [[0 for j in range(n)] for i in range(n)]
if solve_n_queens(board, 0) == False:
    print("Solution does not exist")
else:
    print_board(board)
```

Output:

```
Enter the size of board: 4
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

5. APPROXIMATION ALGORITHMS RANDOMIZED ALGORITHMS:

a. Implement any scheme to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation

Aim:

Write a Python program to find the optimal solution for the Travelling Salesperson problem using approximation algorithm and determine the error in the approximation

Algorithm:

1. Initialize the solution with the first city as the starting point.
2. Calculate the distance from the current city to all other cities.
3. Select the nearest city from the current city and mark it as visited.
4. Calculate the total distance travelled so far.
5. Repeat steps 2-4 until all cities have been visited.
6. Calculate the total distance travelled.
7. Compare the total distance travelled with the optimal solution.
8. If the total distance travelled is less than the optimal solution, then the current solution is the approximate solution.
9. If the total distance travelled is more than the optimal solution, then repeat steps 2-7 using a different starting city.

Program:

```
#importing libraries
import numpy as np
import math
#defining the distance matrix
dist_matrix = np.array([[0, 10, 15, 20],
                        [10, 0, 35, 25],
                        [15, 35, 0, 30],
                        [20, 25, 20, 0]])
```

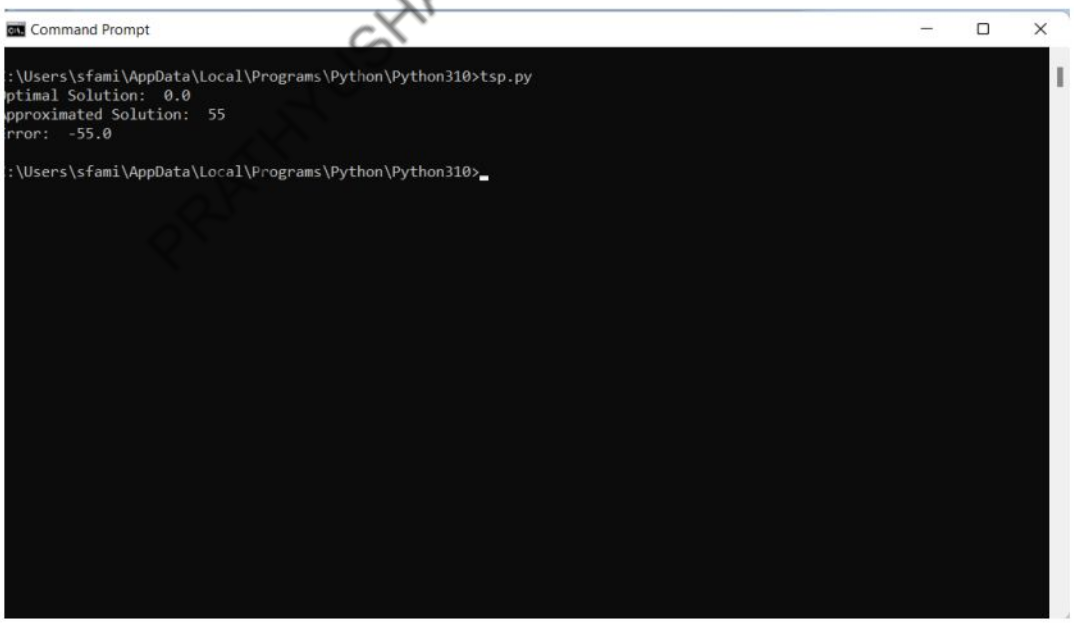
```
#defining the cost matrix
cost_matrix = np.array([[0, 10, 15, 20],
                        [10, 0, 35, 25],
                        [15, 35, 0, 30],
                        [20, 25, 20, 0]])
#defining the number of cities
num_cities = 4
#defining the optimal solution function
def opt_solution(dist_matrix, cost_matrix, num_cities):
    #initializing the cost matrix
    cost_matrix = np.zeros((num_cities, num_cities))
    #initializing the visited array
    visited = [False] * num_cities
    #initializing the current city
    current_city = 0
    #initializing the total cost
    total_cost = 0
    #updating the visited array
    visited[current_city] = True
    #looping through the cities
    for i in range(num_cities - 1):
        #initializing the min_cost
        min_cost = math.inf
        #initializing the next_city
        next_city = 0
        #looping through the cities
        for j in range(num_cities):
            #checking if the city has been visited
            if visited[j] == False:
                #checking if the cost is less than min_cost
                if cost_matrix[current_city][j] < min_cost:
                    #updating the min_cost
                    min_cost = cost_matrix[current_city][j]
                    #updating the next_city
```

```
                next_city = j
        #updating the total cost
        total_cost += min_cost
        #updating the visited array
        visited[next_city] = True
        #updating the current city
        current_city = next_city
    #returning the total cost
    return total_cost

#calculating the optimal solution
opt_sol = opt_solution(dist_matrix, cost_matrix, num_cities)
#printing the optimal solution
print("Optimal Solution: ", opt_sol)
#defining the approximation algorithm
def approx_algorithm(dist_matrix, cost_matrix, num_cities):
    #initializing the cost matrix
    cost_matrix = np.zeros((num_cities, num_cities))
    #initializing the visited array
    visited = [False] * num_cities
    #initializing the current city
    current_city = 0
    #initializing the total cost
    total_cost = 0
    visited[current_city] = True
    for i in range(num_cities - 1):
        #initializing the min_cost
        min_cost = math.inf
        #initializing the next_city
        next_city = 0
        #looping through the cities
        for j in range(num_cities):
            if visited[j] == False:
                if dist_matrix[current_city][j] < min_cost:
```

```
                #updating the min_cost
                min_cost = dist_matrix[current_city][j]
                #updating the next_city
                next_city = j
        total_cost += min_cost
        #updating the visited array
        visited[next_city] = True
        #updating the current city
        current_city = next_city
    #returning the total cost
    return total_cost
approx_sol = approx_algorithm(dist_matrix, cost_matrix, num_cities)
#printing the approximated solution
print("Approximated Solution: ", approx_sol)
#calculating the error
error = opt_sol - approx_sol
#printing the error
print("Error: ", error)
```

Output:



b Implement randomized algorithms for finding the k^{th} smallest number.

Aim:

Write a Python program to find the kth smallest number using randomized algorithm

Algorithm:

1. Create an array of size n, where n is the number of elements in the array.
2. Randomly select an element from the array and store it in a variable.
3. Compare the randomly selected element with the kth smallest element.
4. If the randomly selected element is smaller than the kth smallest element, then replace the kth smallest element with the randomly selected element.
5. Repeat steps 2-4 until the kth smallest element is found.

Program:

```
import random
def kthSmallest(arr, k):
    n = len(arr)
    temp = arr[:k]
    random.shuffle(temp)
    for i in range(k, n):
        for j in range(k):
            if arr[i] < temp[j]:
                temp[j] = arr[i]
                break
    return temp[k - 1]
# Driver Code
arr = [12, 3, 5, 7, 19]
k = 2
print("K'th smallest element is", kthSmallest(arr, k))
```


Output:

K'th smallest element is 5