Ex.No.1. Implement Linear Search. Determine the time required to search foran 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.

Program:

```
def linearSearch(array, n, x):
    # Going through array sequentially
    for i in range(0, n): if
        (array[i] == x):
            return i
    return -1

import matplotlib.pyplot as plt
import time
ot=[]
T=int(input("Enter no of Excution"))
temp=0
y=[]
for j in range(0,T):
    n=int(input("Enter n"))
    y.append(n) my_list=[]
    for i in range(0,n):
        ele=int(input("Enter Element"))
        my_list.append(ele)
    elem_to_search = int(input("Enter key"))
    print("The list is",my_list)
    start=time.time()
    my_result = linearSearch(my_list,n,elem_to_search)
    end=time.time()
    tn=end-start
    if my_result != -1:
        print("Element found at index ", str(my_result))
        else:
        print("Element not found!")
    temp=temp+1
    ot.append(tn)
print("Execution Time",ot)
# Plot a graph
X=ot
Y=y
plt.plot(X,Y)
plt.show()
```

Ex.No.2. 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.

Program:

```
# Python 3 program for recursive binary search.
# Modifications needed for the older Python 2 are found in comments.# Returns
index of x in arr if present, else -1

def binary_search(my_list, low, high, elem):
# Check base case
if high >= low:
        mid = (high + low) // 2
        # If element is present at the middle itself
        if my_list[mid] == elem:
                return mid
        # If element is smaller than mid, then it can only be present in#left subarray
        elif my_list[mid] > elem:
                return binary_search(my_list, low, mid - 1, elem)
        # Else the element can only be present in right subarray
        else:
                return binary_search(my_list, mid + 1, high, elem)
  else:
        # Element is not present in the array
        return -1

# Test array
import matplotlib.pyplot as plt
import time
ot=[]
yvalue=[]
T=int(input("Enter no of Excution"))
for j in range(0,T):
   n=int(input("Enter n"))
   yvalue.append(n)
   my_list=[]
   for i in range(0,n):
      ele=int(input("Enter Element"))
      my_list.append(ele)
   elem_to_search = int(input("Enter key"))
   print("The list is")
   for i in range(0,n):
      print(my_list[i])
   start=time.time()
```

```python
    my_result = binary_search(my_list,0,len(my_list)-1,elem_to_search)
    end=time.time()
    tn=end-start
    if my_result != -1:
        print("Element found at index ", str(my_result))
    else:
        print("Element not found!")
    ot.append(tn)
print(“Execution time”,ot)
# Plot a graph
x=ot
y=yvalue
plt.plot(x,y)
plt.show()
```

Ex. No. 3 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.

Program:

```
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    for i in range(N - M + 1):
        j = 0
        while(j < M):
            if (txt[i + j] != pat[j]):
                break
            j += 1
        if (j == M):
            print("Pattern found at index ", i)
 # Driver's Code
tn=int(input("Enter no. of text"))
txt=[]
for i in range(0,tn):
    ele=str(input("Enter Text Elements"))
    txt.append(ele)
pat=[]
pn=int(input("Enter no. of pattern"))
for j in range(0,pn):
    el=str(input("Enter pattern Elements"))
    pat.append(el)
# Function call
search(pat, txt)
```

Ex. No.4 a Sort a given set of elements using the Insertion 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 take versus n.

Program:

```
def insertion_sort(arr):
     # Traverse through 1 to len(arr)
     for i in range(1, len(arr)):
         key = arr[i]j
         = i - 1
         # Move elements of arr[0..i-1], that are greater than key, to one position ahead of their
  current position
         while j >= 0 and key < arr[j]:
             arr[j + 1] = arr[j]
             j -= 1
         arr[j + 1] = key
#Function Call
 import matplotlib.pyplot as plt
 import time
ot=[]
 T=int(input("Enter no of Excution"))
 temp=0
x=[]
 for j in range(0,T):
     n=int(input("Enter n"))
     x.append(n) my_list=[]
     for i in range(0,n):
         ele=int(input("Enter Element"))
         my_list.append(ele)
     start=time.time()
     my_result = insertion_sort(my_list)
     end=time.time()
     time_comp=end-start
     print("Sorted array is:", my_list)
     temp=temp+1
ot.append(time_comp)
print("Execution Time",ot)
# Plot a graph
X=x
Y=ot
plt.plot(X,Y)
plt.show()
```

Ex. No. 4. b Sort a given set of elements using the 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

Program:

```
# Python program for implementation of heap Sort
# To heapify subtree rooted at index i.
# n is size of heap

def heapify(arr, n, i):
      largest = i  # Initialize largest as root
      l = 2 * i + 1 # left = 2*i + 1
      r = 2 * i + 2 # right = 2*i + 2

   # See if left child of root exists and isgreater than root
      if l < n and arr[i] < arr[l]:
          largest = l
   # See if right child of root exists and is greater than root
      if r < n and arr[largest] < arr[r]:
          largest = r

   # Change root, if needed
   if largest != i:
          (arr[i], arr[largest]) = (arr[largest], arr[i]) # swap

    # Heapify the root.
          heapify(arr, n, largest)

  # The main function to sort an array of given size
  def heapSort(arr):
      n = len(arr)

   # Build a maxheap.
   # Since last parent will be at ((n//2)-1) we can start at that location.
   for i in range(n // 2 - 1, -1, -1):
          heapify(arr, n, i)

   # One by one extract elements
   for i in range(n - 1, 0, -1):
          (arr[i], arr[0]) = (arr[0], arr[i]) # swap
          heapify(arr, i, 0)

  # Driver code to test above
  import matplotlib.pyplot as plt
  import time
  ot=[]
  T=int(input("Enter no of Excution"))
  temp=0
```

```
x=[]
for j in range(0,T):
n=int(input("Enter n"))
x.append(n)
my_list=[]
    for i in range(0,n):
        ele=int(input("Enter Element"))
        my_list.append(ele)
    start=time.time()
    my_result = heapSort(my_list)
    end=time.time()
    time_comp=end-start
    print("Sorted array is:", my_list)
    temp=temp+1
    ot.append(time_comp)
 print("Execution Time",ot)
 # Plot a graph
 X=x
 Y=ot
 plt.plot(X,Y)
 plt.show()
```

Ex. No. 5 Develop a program to implement graph traversal using Breadth First Search

Program:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Ex. No. 6 Develop a program to implement graph traversal using Depth First Search

Program:

```
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Ex. No. 7 Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

Program:

```
# Prim's Algorithm in Python

INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G = [[0, 19, 5, 0, 0],
     [19, 0, 5, 9, 2],
     [5, 5, 0, 1, 6],
     [0, 9, 1, 0, 1],
     [0, 2, 6, 1, 0]]

selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):

    minimum = INF
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```