PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def linear_search(arr, x):
        for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
x=[]
y=[]
for i in range(5):
    n=int(input("\nenter the value of n:"))
    x.append(n)
    arr = [random.randint(0, 1000) for _ in range(n)]
    k=random.randint(0,1000)
    start_time = timer()
    ind=linear_search(arr, k)
    end_time = timer()
    elapsed_time = end_time - start_time
    y.append(elapsed_time)
    print("array elements are in the range of 0-1000")
    print ("k value=",k)
    print("time taken=", elapsed_time)
    print ("element is at the index:",ind)
plt.plot(x,y)
plt.title('Time Taken for Linear Search')
plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.show()
```

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def binary_search(n, a, k, low, high):
    mid = int((low + high) / 2)
  if low > high:
    return -1
  if k == a[mid]:
    return mid
  elif k < a[mid]:
    return binary_search(n, a, k, low, mid - 1)
  else:
    return binary_search(n, a, k, mid + 1, high)
x = []
y = []
for i in range(5):
    n = int(input("\nenter the value of n:"))
  x.append(n)
  arr = [x for x in range(n)]
  k = random.randint(0, n)
  start = timer()
  ind = binary_search(n, arr, k, 0, n - 1)
  end = timer()
  y.append(end - start)
  print("array elements are in the range of 0-",n)
  print("k value=", k)
  print("time taken=", end - start)
  print("element is at the index:", ind)# Plot the results
plt.plot(x, y)
plt.title('Time Taken for Linear Search')
plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.show()
```

PROGRAM:

```
def search(pat,txt):
    m=len(pat)
    n=len(txt)
    for i in range(n-m+1):
        for j in range(m):
            if(txt[i+j]!=pat[j]):
                break
        if(j==m-1):
            print("pattern found at index :",i)
txt=input("enter the text:")
pat=input("enter the pattern to search :")
search(pat,txt)
```

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def insertionSort(array):
    for step in range(1, len(array)):
        key = array[step]
        j = step - 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key
x=[]
y=[]
for i in range(5):
    # Generate a list of random integers
    n=int(input("\nenter the value of n:"))
    x.append(n)
    arr = [random.randint(0, 1000) for _ in range(n)]
    print("\nthe array elements are",arr)
    start_time = timer()
    ind=insertionSort(arr)
    end_time = timer()
    print("array elements are ", arr)
    elapsed_time = end_time - start_time
    y.append(elapsed_time)
    print("time taken=", elapsed_time)
# Plot the results
plt.plot(x,y)
plt.title('Time Taken for insertion sort')
plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.show()
```

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt

def heapify(arr, n, i):

    largest = i

    l = 2 * i + 1

    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:

        largest = l

    if r < n and arr[largest] < arr[r]:

        largest = r

    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)

def heapSort(arr):

    n = len(arr)

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

x=[]

y=[]

for i in range(3):

    n=int(input("\nEnter the value of n:"))

    x.append(n)

    arr = [random.randint(0, 10) for _ in range(n)]

    print("Array elements before sorting are",arr)

    start_time = timer()

    ind=heapSort(arr)

    end_time = timer()

    elapsed_time = end_time - start_time

    y.append(elapsed_time)
```

```
    print("Array elements after sorting are ",arr)
    print("Time taken=", elapsed_time)
plt.plot(x,y)
plt.title('Time Taken for heap sort')
plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.show()
```

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
```

PROGRAM:

```
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')
```

PROGRAM:

```
import sys
vertices = [[0, 0, 1, 1, 0, 0, 0],
            [0, 0, 1, 0, 0, 1, 0],
            [1, 1, 0, 1, 1, 0, 0],
            [1, 0, 1, 0, 0, 0, 1],
            [0, 0, 1, 0, 0, 1, 0],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 0, 1, 0, 1, 0]]
edges = [[0, 0, 1, 2, 0, 0, 0],
         [0, 0, 2, 0, 0, 3, 0],
         [1, 2, 0, 1, 3, 0, 0],
         [2, 0, 1, 0, 0, 0, 1],
         [0, 0, 3, 0, 0, 2, 0],
         [0, 3, 0, 0, 2, 0, 1],
         [0, 0, 0, 1, 0, 1, 0]]
def to_be_visited():
   global visited_and_distance
   v = -10
   for index in range(num_of_vertices):
      if visited_and_distance[index][0] == 0 \and (v < 0 or visited_and_distance[index][1] <= visited_and_distance[v][1]):
         v = index
   return v
num_of_vertices = len(vertices[0])
visited_and_distance = [[0, 0]]
for i in range(num_of_vertices-1):
   visited_and_distance.append([0, sys.maxsize])
for vertex in range(num_of_vertices):
   to_visit = to_be_visited()
   for neighbor_index in range(num_of_vertices):
      if vertices[to_visit][neighbor_index] == 1 and \ visited_and_distance[neighbor_index][0] == 0:
         new_distance = visited_and_distance[to_visit][1] \  + edges[to_visit][neighbor_index]
```

```
            if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance
        visited_and_distance[to_visit][0] = 1

i = 0
for distance in visited_and_distance:
    print("Distance of ", chr(ord('a') + i),
          " from source vertex: ", distance[1])
    i = i + 1
```

PROGRAM:

```
import sys
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])
    def minKey(self, key, mstSet):
        min = sys.maxsize
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v
        return min_index
    def primMST(self):
        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = 0
        mstSet = [False] * self.V
        parent[0] = -1
        for cout in range(self.V):
            u = self.minKey(key, mstSet)
            mstSet[u] = True
            for v in range(self.V):
                if self.graph[u][v] > 0 and mstSet[v] == False \and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u
        self.printMST(parent)
```

```
if __name__ == '__main__':
    g = Graph(5)
    g.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]]
    g.primMST()
```

PROGRAM:

```
V = 4
INF = 99999
def floydWarshall(graph):

    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))

    for k in range(V):

        for i in range(V):
            for j in range(V):
                dist[i][j] = min(dist[i][j],
                                 dist[i][k] + dist[k][j]
                                 )
    printSolution(dist)

def printSolution(dist):
    print("Following matrix shows the shortest distances\
 between every pair of vertices")
    for i in range(V):
        for j in range(V):
            if (dist[i][j] == INF):
                print("%7s" % ("INF"), end=" ")
            else:
                print("%7d\t" % (dist[i][j]), end=' ')
            if j == V - 1:
                print()

if __name__ == "__main__":

    graph = [[0, 5, INF, 10],
             [5, 0, 3, INF],
             [7, INF, 0, 1],
             [INF, INF, 8, 0]
             ]

    floydWarshall(graph)
```

PROGRAM:

```
from collections import defaultdict
class Graph:
    def __init__(self, vertices):
        self.V = vertices

def printSolution(self, reach):
        print("Following matrix transitive closure of the given graph ")
        for i in range(self.V):
            for j in range(self.V):
                if (i == j):
                    print("%7d\t" % (1), end=" ")
                else:
                    print("%7d\t" % (reach[i][j]), end=" ")
            print()

 def transitiveClosure(self, graph):
        reach = [i[:] for i in graph]
        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):
                    reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])
        self.printSolution(reach)

g = Graph(4)
graph = [[1, 1, 0, 1],
         [0, 1, 1, 0],
         [0, 0, 1, 1],
         [0, 0, 0, 1]]
g.transitiveClosure(graph)
```