SEARCHING AND SORTING ALGORITHMS

1. 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: To 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.

ALGORITHM:

STEP 1: Start

STEP 2: Import random,default_timer and matplotlib packages.

STEP 3: Create function def linear_search(arr,x):

Get values of n.

STEP 4: Using random.randint, get random values of n from 0-1000.

STEP 5: Set start_time and end_time .Compute elapsed_time=start_time-end_time.

STEP 6: Initialize ind = linear_search(arr,k)

STEP 7: Plot the graph between n and time using plt.show().

STEP 8: Stop.

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def linear_search(arr, x):
    """Implements linear search algorithm to find an element in a list."""
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
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)]
```

```
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)

# Plot the results

plt.plot(x,y)

plt.title('Time Taken for Linear Search')

plt.xlabel('n')

plt.ylabel('Time (seconds)')

plt.show()
```

RESULT:

Thus the python program to 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 was executed successfully.

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.

AIM:

To write a python program to 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.

ALGORITHM:

STEP 1: Start

STEP 2: Import random,default_timer and matplotlib packages.

STEP 3: def binary_search(n,a,k,low,high):

Mid=int((low+high)/2)

STEP 4: if low>high : return -1.

If k==a[mid]:return mid.

If k<a[mid]: return binay_search(n,a,k,low,mid-1)

End if.

STEP 5: Else : return_search(n,a,k,mid+1,high). End if.

STEP 6: Get the value of n and using random.randint get values of n from 0-1000.

STEP 7: : Set start_time and end_time .Compute elapsed_time=start_time-end_time.

STEP 6: Initialize ind = binary_search(n,arr,k,0,n-1)

STEP 7: Plot the graph between n and time using plt.show().

STEP 8: Stop.

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def binary_search(n, a, k, low, high):
    """Implements binary search algorithm to find an element in a list."""
    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):
    # Generate a list of random integers
    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()
```

RESULT:

Thus the python program to 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 was executed successfully.

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.

AIM:

To write a python program to search a pattern from the given input text and print all occurrences of pattern in that text.

ALGORITHM:

STEP 1: Start

STEP 2: Create function def search(pat,txt):

STEP 3: Initialize m = len (pat) and n=len(txt).

STEP 4: Using for loop, for i in range(n-m+1):

For j in range(m):

If (txt[i+j]!=pat[j]):break. [End if]

STEP 5: if (j==m-1): print("Pattern found at index:",i) [End if]

STEP 6: Get the text and pattern from the user

STEP 7: Define search (pat,txt)

STEP 8: Stop.

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)
```

RESULT:

Thus the python program to search a pattern from the given input text and print all occurrences of pattern in that text was executed successfully.

4. 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.

4A. INSERTION SORT:

AIM:

To write a python program to sort a given set of elements using the Insertion sort 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.

ALGORITHM:

STEP 1: Start

STEP 2: Import random,default_timer and matplotlib packages.

STEP 3: def InsertionSort(arr):

If(n==len(arr))<=1:

Return. [END IF]

STEP 4: for i in range(1,n):

Key=arr[i]

j=i+1

STEP 5: while j=0 and key<arr[j]:

arr[j+1]=arr[j]

j-=1

arr[j+1]=key

STEP 6: Get the value of n and using random.randint get values of n from 0-1000.

STEP 7 : Set start_time and end_time .Compute elapsed_time=start_time-end_time.

STEP 8: Initialize ind = InsertionSort(arr)

STEP 9: Plot the graph between n and time using plt.show().

STEP 10: Stop.

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()
```

RESULT:

Thus the python program to sort a given set of elements using the Insertion sort 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 was executed successfully.

4B. HEAP SORT:

AIM:

To write a python program to sort a given set of elements using the heap sort 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.

ALGORITHM:

STEP 1: Start

STEP 2: Import random,default_timer and matplotlib packages.

STEP 3: Create def heapify(arr,n,i)

And write the necessary conditions to construct the min or max heap by heapify(arr,n,largest)

STEP 4: Using for loop, for i in range(3):

Get the value of n.

STEP 5: Get random values from 0 to 10 in the given range of n.

STEP 6: Print the array before heap sort.

STEP 7 : Set start_time and end_time .Compute elapsed_time=start_time-end_time.

STEP 8: Now Print the array after sorting.

STEP 9: Print the time taken for the sorting.

STEP 10: Plot the graph between n and time using plt.show().

STEP 11: Stop.

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()
```

RESULT:

Thus the python program to sort a given set of elements using the heap sort 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 was executed successfully.

GRAPH ALGORITHMS

1. Develop a program to implement graph traversal using Breadth First Search

AIM:

To write a python program to implement graph traversal using Breadth First Search.

ALGORITHM:

STEP 1: Start

STEP 2: Give the input graph.

STEP 3: Create empty lists for visited nodes and queue.

STEP 4: Create a function, def bfs(visited,graph,node)

STEP 5: Append the traversal node to the visited and queue.

STEP 6: Using while loop, while queue: m=queue.pop(0).Print m. [End while loop]

STEP 7: Using for loop ,for neighbor in graph[m]:

If neighbor is not visited, then append neighbors to the visited and queue list. [End if] [End for loop]

STEP 8: Call the function.

STEP 9: Stop.

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
```

RESULT:

Thus the python program to implement graph traversal using Breadth First Search was executed successfully.

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

AIM:

To write a python program to implement graph traversal using Depth First Search.

ALGORITHM:

STEP 1: Start

STEP 2: Give the input graph.

STEP 3: Define the visited list as set().

STEP 4: Create a function, def dfs(visited,graph,node)

STEP 5: Using if loop ,if node not in visited: Print node. Add the node to the visited list.

STEP 6: Using for loop ,for neighbor in graph[node]:

dfs(visited,graph,neighbor) [End for loop]

STEP 7: Call the function.

STEP 8: Stop.

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')
```

RESULT:

Thus the python program to implement graph traversal using Depth First Search was executed successfully.

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

AIM:

To develop a python program to find the shortest paths to other vertices using Dijkstra's algorithm in a weighted connected graph.

ALGORITHM:

STEP 1: Start

STEP 2: Import sys

STEP 3: Provide the graph of vertices and edges.

STEP 4: To find which vertex is to be visited next, create a function, def to_be_visited():

Declare global as visited_and_distance and v=-10.

STEP 5: Return the visited vertices using for loop giving range as num_of_vertices.

STEP 6: Assign num_of_vertices=len(vertices[0]) and visited_and_distance=[[0,0]].

STEP 7: Find the shortest distance from the source vertex and print them.

STEP 8: Stop.

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
```

RESULT:

Thus the python program to find the shortest paths to other vertices using Dijkstra's algorithm in a weighted connected graph was executed successfully.

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

AIM:

To write a python program to find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

ALGORITHM:

STEP 1: Start

Step 2: Determine an arbitrary vertex as the starting vertex of the MST.

Step 3: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).

Step 4: Find edges connecting any tree vertex with the fringe vertices.

Step 5: Find the minimum among these edges.

Step 6: Add the chosen edge to the MST if it does not form any cycle.

Step 7: Return the MST .

STEP 8: Stop.

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()
```

RESULT:

Thus the python program to find the minimum cost spanning tree of a given undirected graph using Prim's algorithm was executed successfully.

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

AIM:

To write a python program to implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem.

ALGORITHM:

STEP 1: Start

Step 2: Define V=4,INF=99999,define floydWarshall(graph):

Step 3: Define dist and printSolution(dist)

Step 4: Define function def printSolution(dist)

If(dist[i][j]==INF) then print INF

Else print dist[i][j]

Step 5: if j==v-1 then print()

Step 6: Inside main give the input graph and call the function floydWarshall(graph)

STEP 7: Stop.

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)
```

RESULT:

Thus the python program to implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem was executed successfully.

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

AIM:

To write a python program to compute the transitive closure of a given directed graph using Warshall's algorithm.

ALGORITHM:

STEP 1: Start

Step 2: from collections import defaultdict and define class Graph

Step 3: Initialize self.V=vertices and printSolution(self,reach)

Step 4: if(i==j) then print 1

Else print reach[i][j].

Step 5: Define transitiveClosure(self,graph)

Self.printSolution(reach)

Step 6: Initialize g=Graph(4).Give the input graph.

Step 7: Call transitiveClosure(graph).

STEP 8: Stop.

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)
```

RESULT:

Thus the python program to compute the transitive closure of a given directed graph using Warshall's algorithm was executed successfully.

ALGORITHM DESIGN TECHNIQUES

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

AIM:

To write a python program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.

ALGORITHM:

STEP 1: Start

Step 2: Define def find_max_min(arr) and n=len(arr).

Step 3: If n==1

Return arr[0],arr[0]

Step 4: elif n==2

Return (arr[0],arr[1])

if arr[0]<arr[1]
else(arr[1],arr[0])

Step 5 : else:

mid=n//2

Step 6: Find the middle element of the array and split the array into two halves.

Step 7: Find the min and max for left sublist and right sublist.

Step 8: Define the list and find min and max by find_max_min(arr).

Step 9: Stop.

PROGRAM:

```
def find_max_min(arr):
    n=len(arr)
    if n==1:
        return arr[0],arr[0]
    elif n==2:
            return (arr[0],arr[1]) if arr[0]<arr[1] else(arr[1],arr[0])

    else:
        mid=n//2
        left_min,left_max=find_max_min(arr[:mid])
        right_min,right_max=find_max_min(arr[mid:])
        return min(left_min,right_min),max(left_max,right_max)
arr=[3,5,1,78,56,94,23,45]
mini,maxi=find_max_min(arr)
print("The max no is:",maxi)
print("The min no is:",mini)
```

RESULT:

Thus the python program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique was executed successfully.

2. 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 in the list to be sorted and plot a graph of the time taken versus n.

A.MERGE SORT:

AIM:

To write a python program to perform merge sort an array of elements and determine the time required to sort. 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.

ALGORITHM:

STEP 1: Start

STEP 2: Import random,default_timer and matplotlib packages.

STEP 3: Create function def mergeSort(array):

STEP 4: Initially divide the array into 2 equal halves.

STEP 5: Divide the array until we cannot divide it anymore.

STEP 6: Now sort the divided parts one by one.

STEP 7: Using random.randint, get random values of n from 0-1000.

STEP 8: Set start_time and end_time .Compute elapsed_time=start_time-end_time.

STEP 9: Initialize ind = mergeSort(array)

STEP 10: Plot the graph between n and time using plt.show().

STEP 11: Stop.

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def mergeSort(array):
    if len(array) > 1:
        r = len(array)//2
        L = array[:r]
        M = array[r:]
        mergeSort(L)
        mergeSort(M)
        i = j = k = 0
        while i < len(L) and j < len(M):
            if L[i] < M[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = M[j]
                j += 1
            k += 1
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1
        while j < len(M):
```

```
            array[k] = M[j]
            j += 1
            k += 1
x=[]
y=[]
for i in range(3):
    # 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=mergeSort(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 merge sort')
plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.show()
```

RESULT:

Thus the python program to perform merge sort an array of elements and determine the time required to sort. 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 was executed successfully.

B.QUICK SORT:

AIM:

To write a python program to perform quick sort an array of elements and determine the time required to sort. 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.

ALGORITHM:

STEP 1: Start

Import random,default_timer and matplotlib packages.

STEP 3: Create function def partition(array,low,high):

Select the last element of array as pivot element.

STEP 4: Define def quicksort(array,low,high):

Implement partition and perform quick sort.

STEP 5: Using random.randint, get random values of n from 0-1000.

STEP 6: Set start_time and end_time .Compute elapsed_time=start_time-end_time.

STEP 7: Initialize ind = quickSort(array)

STEP 8: Plot the graph between n and time using plt.show().

STEP 9: Stop.

PROGRAM:

```
import random
from timeit import default_timer as timer
import matplotlib.pyplot as plt
def partition(array, low, high):
  pivot = array[high]
  i = low - 1
  for j in range(low, high):
    if array[j] <= pivot:
      i = i + 1
      (array[i], array[j]) = (array[j], array[i])
  (array[i + 1], array[high]) = (array[high], array[i + 1])
  return i + 1
def quickSort(array, low, high):
  if low < high:
    pi = partition(array, low, high)
    quickSort(array, low, pi - 1)
    quickSort(array, pi + 1, high)
x=[]
y=[]
for i in range(3):
    # 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=quickSort(arr,low=0,high=n-1)
    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 quick sort')
plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.show()
```

RESULT:

Thus the python program to perform quick sort an array of elements and determine the time required to sort. 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 was executed successfully.

STATE SPACE SEARCH ALGORITHMS

1. Implement N Queens problem using Backtracking.

AIM:

To write a python program to Implement N Queens problem using Backtracking.

ALGORITHM:

STEP 1: Start

Step 2:

Step 3:

Step 4:

Step 5:

Step 6:

Step 7:

STEP 8: Stop.

PROGRAM:

RESULT:

Thus the python program to implement N Queens problem using Backtracking was executed successfully.

APPROXIMATION ALGORITHMS RANDOMIZED ALGORITHMS

1. 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:

To write a python program to 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.

ALGORITHM:

STEP 1: Start

Step 2:

Step 3:

Step 4:

Step 5:

Step 6:

Step 7:

STEP 8: Stop.

PROGRAM:

RESULT:

Thus the python program to 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 was executed successfully.

2. Implement randomized algorithms for finding the kth smallest number.

AIM:

To write a python program to implement randomized algorithms for finding the kth smallest number.

ALGORITHM:

STEP 1: Start

Step 2:

Step 3:

Step 4:

Step 5:

Step 6:

Step 7:

STEP 8: Stop.

PROGRAM:

RESULT:

Thus the python program to implement randomized algorithms for finding the kth smallest number was executed successfully.