R P Sarathy Institute of Technology

ACADEMIC YEAR-2020-2024

EVEN SEMESTER

LAB MANUAL

REGULATION- 2021

CS3401- ALGORITHMS

EXP.NO:1 **IMPLEMENTATION OF LINEAR SEARCH**

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 :

1. Declare an array.
2. The linear_search function takes an array arr and an element x as input, and searches for the element in the array using linear search.
3. If the element is found, it returns the index of the element in the array. Otherwise, it returns -1.
4. The program defines a list n_values containing different values of n to test the linear search algorithm on.
5. It then loops through this list, generates a random list of n elements, and searches for a random element in the list.
6. It measures the time taken to perform the search using the time module, and appends the time taken to a list time_values.
7. Finally, the program uses matplotlib library to plot a graph of the time takenversus n.

PROGRAM:

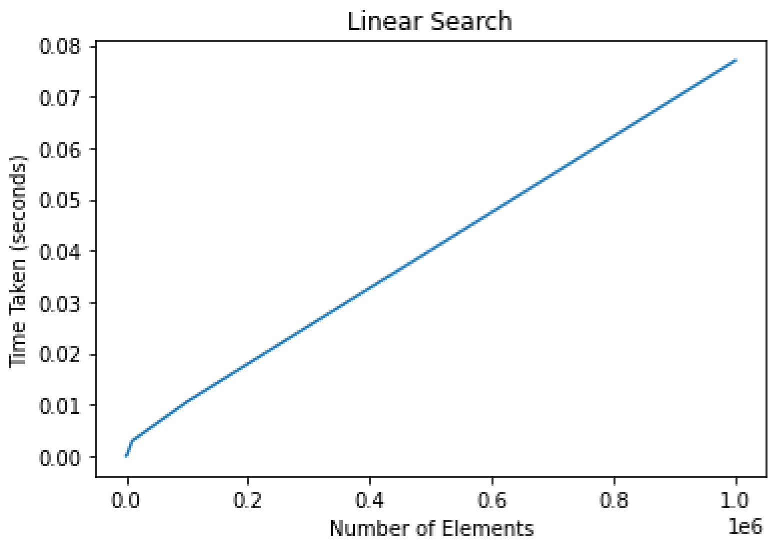
```
import time
import matplotlib.pyplot as plt
import random

def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

n_values = [100, 1000, 10000, 100000, 1000000]
time_values = []

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

    start_time = time.time()
```

```
    linear_search(arr, x)
    end_time = time.time()

    time_values.append(end_time - start_time)

plt.plot(n_values, time_values)
plt.title('Linear Search')
plt.xlabel('Number of Elements')
plt.ylabel('Time Taken (seconds)')
plt.show()
```

OUTPUT:

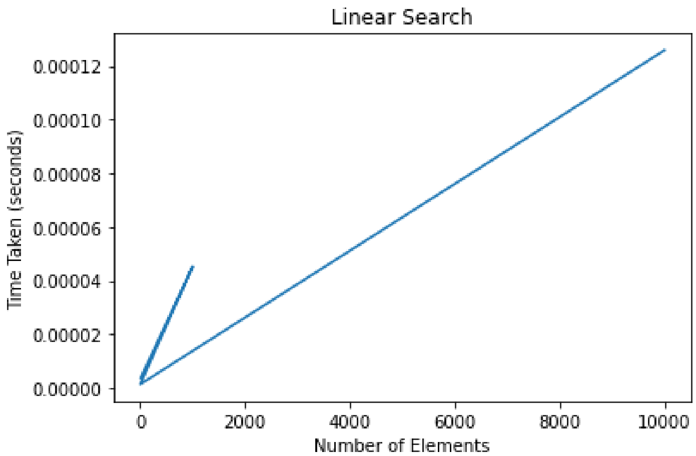
Output 1:

n_values = [100, 1000, 10000, 100000, 1000000]



Output 2 :

n_values = [10, 100, 1000, 1, 10000]



Result: Thus the python program for implementation of linear search program was executed and verified successfully.

EXP.NO:2 **IMPLEMENTATION OF RECURSIVE BINARY SEARCH**

AIM :

To Implement recursive Binary Search. Determine the time required to searchan 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 :

1. Declare the array.
2. '**binary_search_recursive**' is a recursive function that takes an array '**arr**', the lower and upper bounds of the subarray being searched **'low** '**and** **'high'**, and the element being searched for '**x**'.
3. It returns the index of the elementif it is found, or -1 if it is not found.
4. The function '**test_binary_search_recursive**' generates arrays of different sizes and runs a binary search for a random element in each array.
5. It records the timetaken to run the search and plots it on a graph.
6. The graph shows the time taken to search for an element versus the size of the array being searched.
7. As the size of the array increases, the time taken to searchfor an element increases as well, but the increase is logarithmic since binary search has a time complexity of O(log n).

PROGRAM :

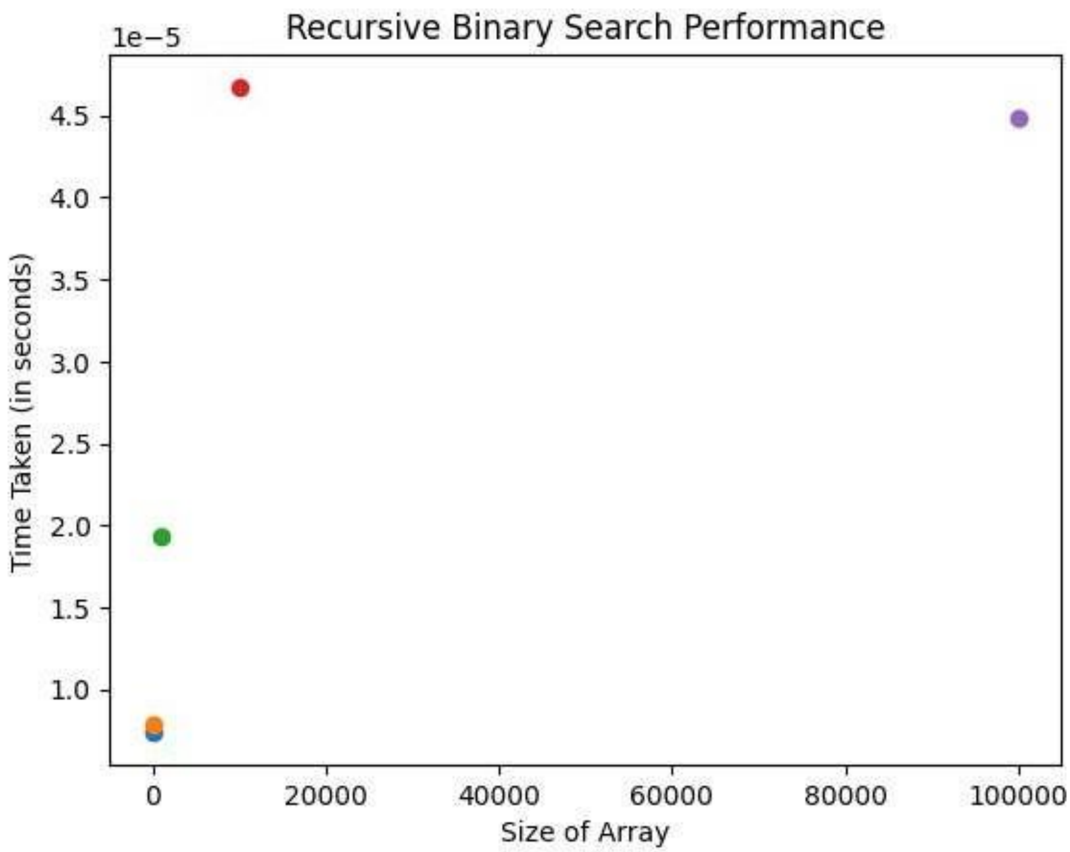
```
import random
import time
import matplotlib.pyplot as plt

def binary_search_recursive(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2

        if arr[mid] == x:
            return mid

        elif arr[mid] > x:
            return binary_search_recursive(arr, low, mid - 1, x)

        else:
            return binary_search_recursive(arr, mid + 1, high, x)
```

```
    else:
        return -1


def test_binary_search_recursive():
    for n in [10, 100, 1000, 10000, 100000]:
        arr = [random.randint(1, n) for i in range(n)]
        arr.sort()

        start_time = time.time()
        x = random.randint(1, n)
        result = binary_search_recursive(arr, 0, n-1, x)
        end_time = time.time()

        if result == -1:
            print(f"Element {x} not found in the array")
        else:
            print(f"Element {x} found at index {result}")

        print(f"Time taken to search in array of size {n}: {end_ti
me - start_time}")
        print("=" * 50)

        plt.scatter(n, end_time - start_time)

    plt.title("Recursive Binary Search Performance")
    plt.xlabel("Size of Array")
    plt.ylabel("Time Taken (in seconds)")
    plt.show()

test_binary_search_recursive()
```

OUTPUT:

```
Element 4 not found in the array
Time taken to search in array of size 10: 7.3909759521484375e-06
==================================================
Element 31 found at index 36
Time taken to search in array of size 100: 7.867813110351562e-06
==================================================
Element 414 found at index 393
Time taken to search in array of size 1000: 1.9311904907226562e-05
```

```
==================================================
Element 4378 not found in the array
Time taken to search in array of size 10000: 4.673004150390625e-05
==================================================
Element 52551 found at index 52435
Time taken to search in array of size 100000: 4.482269287109375e-05
 =======================================
```



Result:

Thus the python program for implementation of recursive binary search was executed and verified successfully.

EXP.NO: 3 **PATTERN MATCHING**

AIM :

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

ALGORITHM :

1. One way to implement the search function is to use the brute-force approach, which involves comparing each possible substring of the text with the pattern.

2. The algorithm iterates through the text from the first character to the (n-m)th character and checks whether the pattern matches the substring of the text starting at that position.

3. If a match is found, the function prints the index of the match.

PROGRAM:

```
def search(pat, txt):
    n = len(txt)
    m = len(pat)
    result = []

    # Loop through the text and search for the pattern
    for i in range(n-m+1):
        j = 0
        while(j < m):
            if (txt[i+j] != pat[j]):
                break
            j += 1

        # If the entire pattern is found, add the index to the res
ult list
        if (j == m):
            result. append(i)

    return result
txt = "AABAACAADAABAABA"
pat = "AABA"
result = search(pat, txt)
print("Pattern found at indices:", result)
```

OUTPUT:

Pattern found at indices: [0, 9, 12]

Result: Thus the python program implementation of pattern matching was executed and verified successfully.

EXP.NO: 4 IMPLEMENTATION OF INSERTION SORT AND HEAP SORT

AIM :

To 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 fordifferent values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

ALGORITHM:

Algorithm for insertion sort:

1. The **insertionSort** function takes a list of elements and sorts them using the Insertion sort algorithm.

2.The **generateList** function generates a list of **n** random numbers between 1 and 1000.

3.The **measureTime** function generates a list of **n** random numbers, sorts it using the **insertionSort** function, and measures the time required to sort the list.

4.The **plotGraph** function generates a list of **n** values and calls the **measureTime** function for each **n** value. It then plots a graph of the time required to sort the list versus the value of **n**.

Algorithm for heap sort:

1. The **heapify** function takes an array **arr**, the size of the heap **n**, and the root index **i** of the subtree to heapify. It compares the root node with its left and right children and swaps the root with the larger child if necessary. The function then recursively calls itself on the subtree with the new root index.

2. The **heapSort** function takes an array **arr** and sorts it using the Heap sort algorithm. It first builds a max heap by heapifying all subtrees bottom-up. It then repeatedly extracts the maximum element from the heap and places it at the end of the array.

3. The **generateList** function generates a list of **n** random numbers between 1 and 1000.

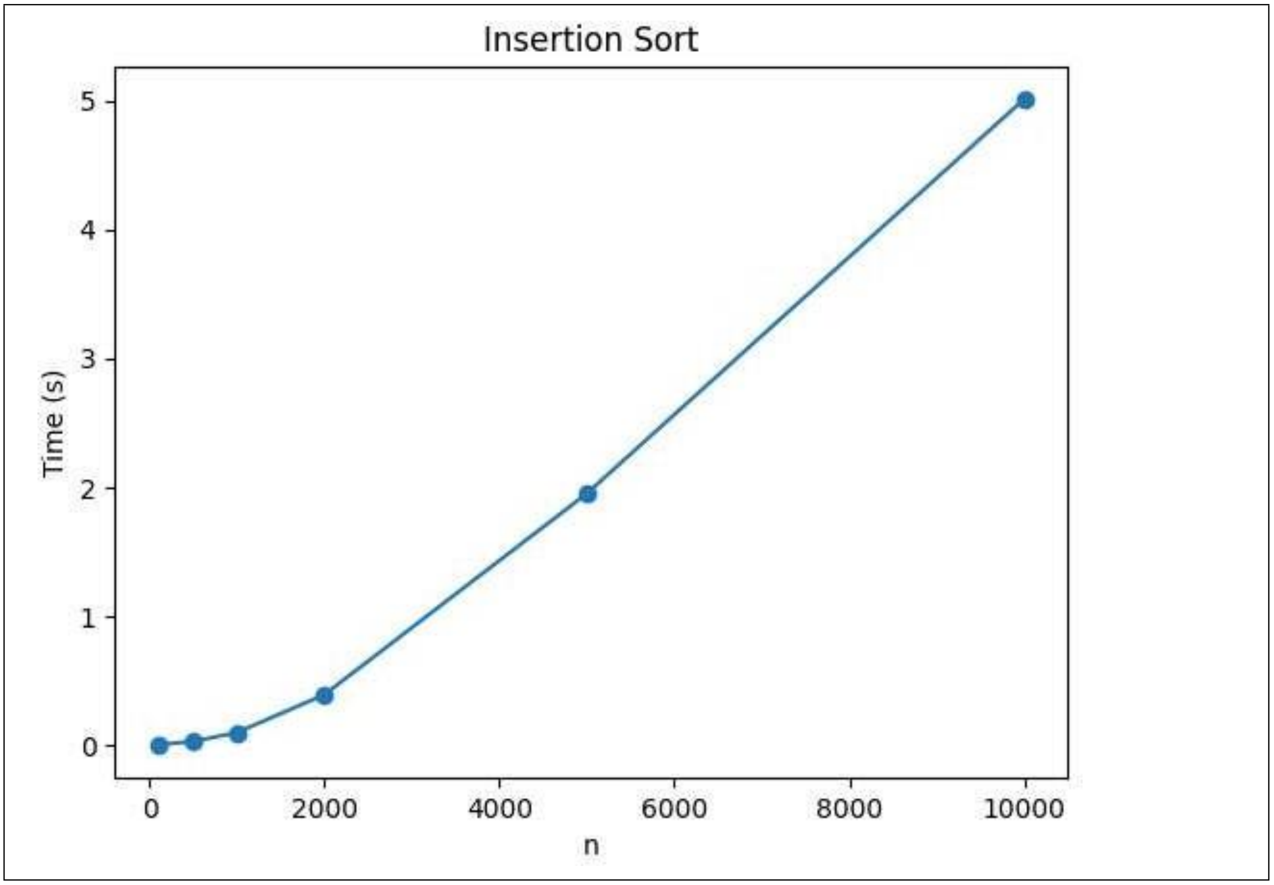
4. The **measureTime** function generates a list of **n** random numbers, sorts it using the **heapSort** function, and measures the time required to sort the list.

5. The **plotGraph** function generates a list of **n** values and calls the **measureTime** function for each **n** value. It then plots a graph of the time required to sort the list versus the value of **n**.

PROGRAM:INSERTION SORT

```
import matplotlib.pyplot as plt
import random
import time

def insertionSort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Generate a list of n random numbers
def generateList(n):
    return [random.randint(1, 1000) for i in range(n)]

# Measure the time required to sort a list of n elements
def measureTime(n):
    arr = generateList(n)
    startTime = time.time()
    insertionSort(arr)
    endTime = time.time()
    return endTime - startTime

# Plot a graph of the time required to sort a list of n elements
def plotGraph(nList):
    timeList = [measureTime(n) for n in nList]
    plt.plot(nList, timeList, 'o-')
    plt.xlabel('n')
    plt.ylabel('Time (s)')
    plt.title('Insertion Sort')
    plt.show()
nList = [100, 500, 1000, 2000, 5000, 10000]
plotGraph(nList)
```

OUTPUT:

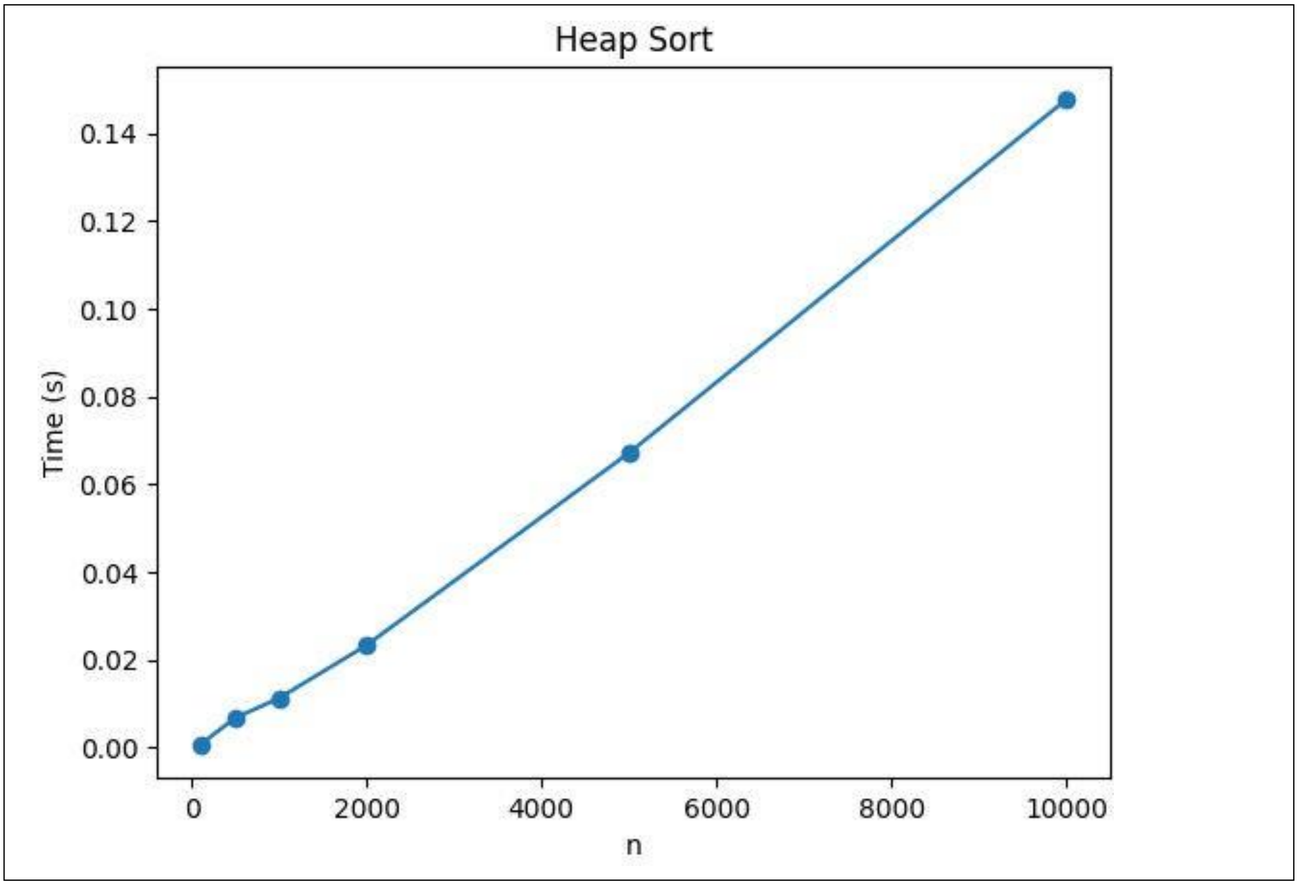


PROGRAM:HEAPSORT

```
import matplotlib.pyplot as plt
import random
import time

# Heapify a subtree rooted with node i
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1 # left child
    r = 2 * i + 2 # right child

    # See if left child of root exists and is greater 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)

# Heap sort function
def heapSort(arr):
    n = len(arr)

    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

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

# Generate a list of n random numbers
def generateList(n):
    return [random.randint(1, 1000) for i in range(n)]

# Measure the time required to sort a list of n elements
def measureTime(n):
    arr = generateList(n)
    startTime = time.time()
    heapSort(arr)
    endTime = time.time()
    return endTime - startTime

# Plot a graph of the time required to sort a list of n elements
def plotGraph(nList):
    timeList = [measureTime(n) for n in nList]
    plt.plot(nList, timeList, 'o-')
    plt.xlabel('n')
    plt.ylabel('Time (s)')
    plt.title('Heap Sort')
    plt.show()
nList = [100, 500, 1000, 2000, 5000, 10000]
plotGraph(nList)
```

OUTPUT:



RESULT: Thus the . python program for implementation of insertion sort and heap sort was executed and verified successfully.

EXP.NO: 5 IMPLEMENTATION OF GRAPH TRAVERSAL USING BREADTH FIRST SEARCH

AIM :

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

ALGORITHM:

Step 1:Start by putting any one of the graph's vertices at the back of a queue.

Step 2:Take the front item of the queue and add it to the visited list.

Step 3:Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Step 4:Keep repeating steps 2 and 3 until the queue is empty.

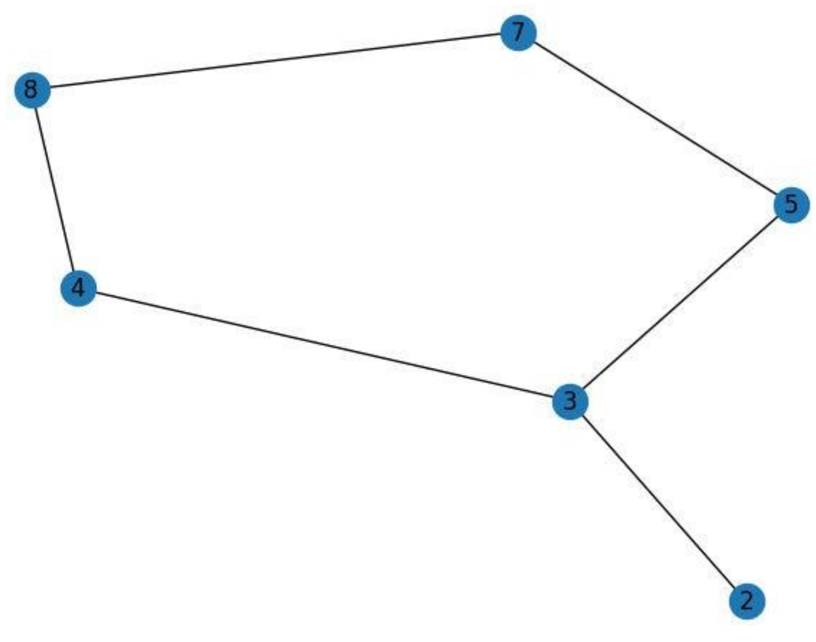
PROGRAM:

```
import networkx as nx
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
G = nx.Graph(graph)
nx.draw(G, with_labels = True)
visited = [] # List for visited nodes.
queue = []      #Initialize a queue

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

# Creating loop to visit each node
  while queue:
    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
```

OUTPUT:

```
Following is the Breadth-First Search
5 3 7 2 4 8
```



RESULT: Thus the python program for implementation of graph traversal using breadth first search was executed and verified successfully.

EXP.NO: 6 IMPLEMENTATION OF GRAPH TRAVERSAL USING DEPTH FIRST SEARCH

AIM :

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

ALGORITHM :

Step 1: Start by putting any one of the graph's vertices on top of a stack.

Step 2:Take the top item of the stack and add it to the visited list.

Step 3:Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

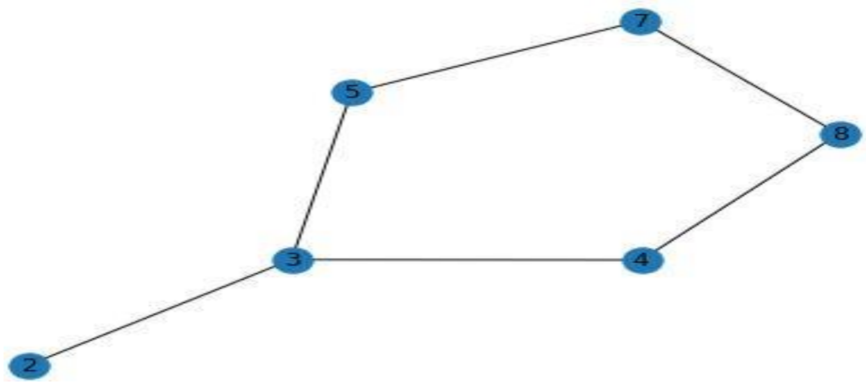
Step 4:Keep repeating steps 2 and 3 until the stack is empty.

PROGRAM :

```
# Using adjacency list
g = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
G = nx.Graph(g)
nx.draw(G, with_labels = True)
visited = set()
# Set to keep track of visited nodes of graph.

def dfs(visited, g, node): dfs
if  node not in visited:
  print (node)
  visited.add(node)
  for neighbour in g[node]:
    dfs(visited, g, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, g, '5')
```

OUTPUT:

```
Following is the Depth-First Search
5
3
2
4
8
7
```



RESULT: Thus the python program for implementation of graph traversal using breadth first search was executed and verified successfully.

EXP.NO: 7 IMPLEMENTATION OF DIJIKSTRA'S ALGORITHM

AIM: To develop a program to find the shortest paths to other vertices using Dijkstra's algorithm.

ALGORITHM :

1. First, we define a function 'dijkstra' that takes three arguments: the graph represented as an adjacency matrix, the starting vertex src, and the number of vertices in the graph n.

2. The function returns a list of shortest distances from the source vertex to all other vertices in the graph.

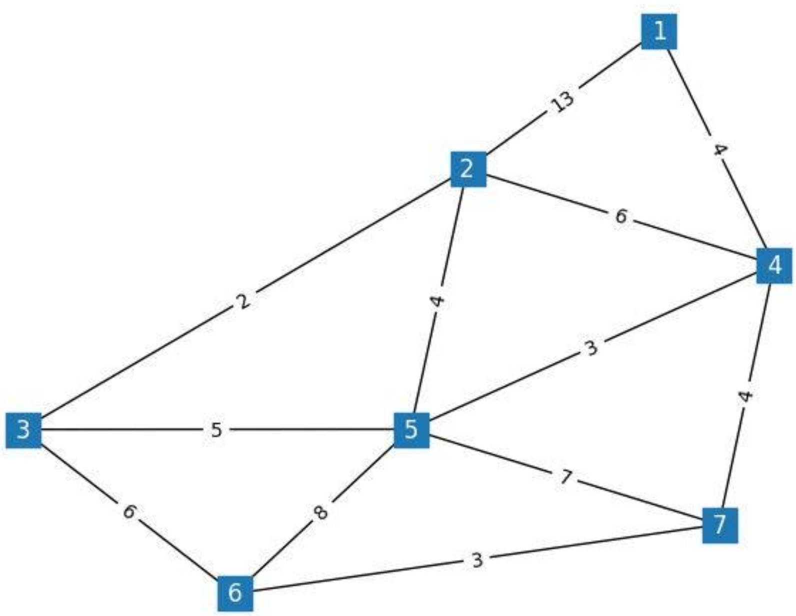
PROGRAM:

```
# importing network
import networkx as nx
import pylab
import matplotlib.pyplot as plt

# Create an empty Undirected Weighted Graph
G = nx.Graph()

nodes_list = [1, 2, 3, 4, 5, 6, 7]
G.add_nodes_from(nodes_list)

# Add weighted edges
edges_list = [(1, 2, 13),(1, 4, 4),(2, 3, 2),(2, 4, 6), (2, 5, 4),
 (3, 5, 5),
              (3, 6, 6),(4, 5, 3),(4, 7, 4), (5, 6, 8), (5, 7, 7),
 (6, 7, 3)]
G.add_weighted_edges_from(edges_list)

plt.figure()
pos = nx.spring_layout(G)
weight_labels = nx.get_edge_attributes(G,'weight')
nx.draw(G,pos,font_color = 'white', node_shape = 's', with_labels
= True,)
nx.draw_networkx_edge_labels(G,pos,edge_labels=weight_labels)

pos = nx.planar_layout(G)

#Give us the shortest paths from node 1 using the weights from the
 edges.
p1 = nx.shortest_path(G, source=1, weight="weight")
```

```
# This will give us the shortest path from node 1 to node 6.
p1to6 = nx.shortest_path(G, source=1, target=6, weight="weight")

# This will give us the length of the shortest path from node 1 to
 node 6.
length = nx.shortest_path_length(G, source=1, target=6, weight="we
ight")

print("All shortest paths from 1: ", p1)
print("Shortest path from 1 to 6: ", p1to6)
print("Length of the shortest path: ", length)
```

OUTPUT:

```
All shortest paths from 1:  {1: [1], 2: [1, 4, 2], 4: [1, 4], 5: [1, 4, 5], 7:
[1, 4, 7], 3: [1, 4, 5, 3], 6: [1, 4, 7, 6]}
Shortest path from 1 to 6:   [1, 4, 7, 6]
Length of the shortest path:   11
```



RESULT: Thus the python program to find the shortest paths to other vertices using Dijkstra's algorithm was executed and verified successfully.

EX.NO:8 IMPLEMENTATION OF PRIM'S ALGORITHM

AIM:

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

ALGORITHM :

Step 1: Determine the arbitrary starting vertex.

Step 2: Keep repeating steps 3 and 4 until the fringe vertices (vertices not included in MST) remain.

Step 3: Select an edge connecting the tree vertex and fringe vertex having the minimum weight.

Step 4: Add the chosen edge to MST if it doesn't form any closed cycle.

Step 5: Exit

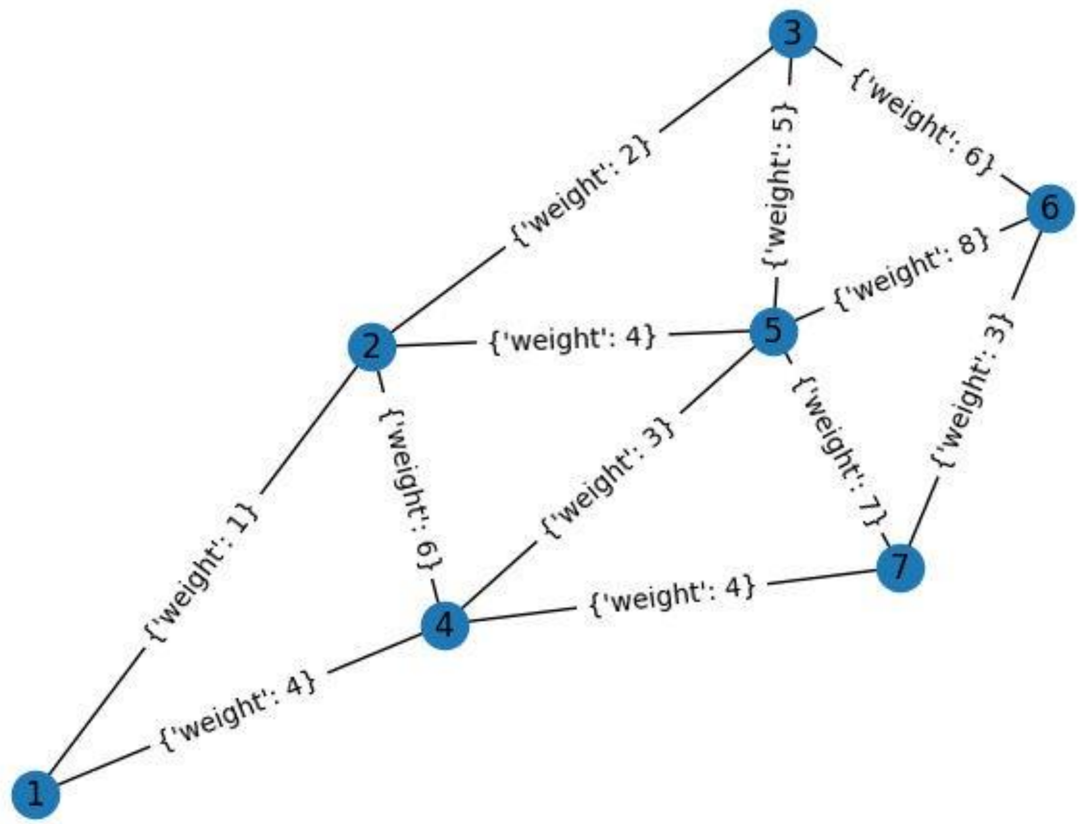
PROGRAM :

```
import matplotlib.pyplot as plt
import networkx as nx
import pylab


# Create an empty Undirected Weighted Graph
G = nx.Graph()

# Add nodes
nodes_list = [1, 2, 3, 4, 5,   6, 7]
G.add_nodes_from(nodes_list)

# Add weighted edges
edges_list = [(1, 2, 1), (1,  4, 4), (2, 3, 2), (2, 4, 6), (2, 5, 4
), (3, 5, 5),
              (3, 6, 6), (4, 5, 3), (4, 7, 4), (5, 6, 8), (5, 7, 7
), (6, 7, 3)]
G.add_weighted_edges_from(edges_list)

pos=nx.spring_layout(G)
pylab.figure(1)
nx.draw(G,pos, with_labels= 'true')
# use default edge labels
nx.draw_networkx_edge_labels(G,pos)
```

```
# Calculate a minimum spanning tree of an undirected weighted grap
h with
# the Prim algorithm
mst = nx.minimum_spanning_tree(G, algorithm='prim')
print(sorted(mst.edges(data=True)))
```

OUTPUT:

```
[(1, 2, {'weight': 1}), (1, 4, {'weight': 4}), (2, 3, {'weight': 2}), (4, 5,
{'weight': 3}), (4, 7, {'weight': 4}), (6, 7, {'weight': 3})]
```



RESULT: Thus the python program for implementation of minimum cost spanning tree of a given undirected graph using Prim's algorithm.

EX.NO:9 IMPLEMENTATION OF FLOYD'S ALGORITHM FOR THE ALL-PAIRS-SHORTEST- PATHS PROBLEM

AIM: To Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem.

ALGORITHM:

Step1: In this program, **INF** represents infinity, and the **floyd_algorithm** function takes in a weighted graph represented as a two-dimensional list where **graph[i][j]** is the weight of the edge from vertex **i** to vertex **j**.

Step:2 The function returns a two-dimensional list **dist** where **dist[i][j]** is the shortest path from vertex **i** to vertex **j**.

Step:3 The algorithm first initializes the **dist** list with the weights of the edges in the graph. It then uses three nested loops to find the shortest path from vertex **i** to vertex **j** through vertex **k**.

Step:4 If the path through **k** is shorter than the current shortest path from **i** to **j**, it updates **dist[i][j]** with the new shortest path.

Step:5 Finally, the program calls the **floyd_algorithm** function on a sample input graph and prints the resulting **dist** list.

PROGRAM:

```
INF = float('inf')

def floyd_algorithm(graph):
    n = len(graph)
    dist = [[INF for j in range(n)] for i in range(n)]

    for i in range(n):
        for j in range(n):
            if graph[i][j] != 0:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
```

```
    return dist

# Sample input
graph = [
    [0, 5, INF, 10],
    [INF, 0, 3, INF],
    [INF, INF, 0, 1],
    [INF, INF, INF, 0]
]

# Run the algorithm and print the result
result = floyd_algorithm(graph)
for row in result:
    print(row)
```

OUTPUT:

```
[inf, 5, 8, 9]
[inf, inf, 3, 4]
[inf, inf, inf, 1]
[inf, inf, inf, inf]
```

RESULT: Thus the python program for implementation of Floyd's algorithm for the All-Pairs- Shortest-Paths problem was executed and verified successfully.

EX.NO:10 COMPUTE THE TRANSITIVE CLOSURE OF A DIRECTED GRAPH USING WARSHALL'S ALGORITHM

AIM: To Compute the transitive closure of a given directed graph using Warshall's algorithm.

ALGORITHM:

Step1: In this program, graph is a two-dimensional list representing the directed graph where graph[i][j] is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.

Step2: The warshall_algorithm function returns a two-dimensional list representing the transitive closure of the input graph.

Step3: The algorithm first creates a copy of the input graph as the initial transitive closure. It then uses three nested loops to update the transitive closure by checking if there is a path from vertex i to vertex j through vertex k. If there is, it sets transitive_closure[i][j] to 1.

Step4: Finally, the program calls the warshall_algorithm function on a sample input graph and prints the resulting transitive closure.

PROGRAM:

```
def warshall_algorithm(graph):
    n = len(graph)

    # Create a copy of the original graph
    transitive_closure = [row[:] for row in graph]

    # Compute the transitive closure using Warshall's algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                transitive_closure[i][j] = transitive_closure[i][j
] or (transitive_closure[i][k] and transitive_closure[k][j])

    return transitive_closure

# Sample input
graph = [
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1],
```

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

# Run the algorithm and print the result
result = warshall_algorithm(graph)
for row in result:
    print(row)
```

OUTPUT:

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

RESULT: Thus the python program to Compute the transitive closure of a given directed graph using Warshall's algorithm was executed and verified successfully.

EX.NO:11 IMPLEMENTATION OF FINDING THE MAXIMUM AND MINIMUM NUMBERS IN A IST USING DIVIDE AND CONQUER TECHNIQUE

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

ALGORITHM:

Step1: The **find_max_min** function recursively divides the list into two halves until the base cases are reached (when the list contains only one or two elements).

Step2: In the base case, the maximum and minimum numbers are returned.

Step3: In the recursive case, the maximum and minimum numbers of the left and right halves are computed and the maximum and minimum of the whole list is returned using the **max** and **min** functions.

PROGRAM:

```
def find_max_min(arr):
    if len(arr) == 1:
        return arr[0], arr[0]
    elif len(arr) == 2:
        if arr[0] > arr[1]:
            return arr[0], arr[1]
        else:
            return arr[1], arr[0]
    else:
        mid = len(arr) // 2
        left_max, left_min = find_max_min(arr[:mid])
        right_max, right_min = find_max_min(arr[mid:])
        return max(left_max, right_max), min(left_min, right_min)

# Example usage
arr = [3, 1, 5, 2, 9, 7]
max_num, min_num = find_max_min(arr)
print("Maximum number:", max_num)
print("Minimum number:", min_num)
```

OUTPUT:

```
Maximum number: 9
Minimum number: 1
```

RESULT: Thus the python program for find out the maximum and minimum numbers in a given list of *n* numbersusing the divide and conquer technique was executed and verified successfully.

EX.NO:12(A) IMPLEMENTATION OF MERGE SORT

AIM: To Implement Merge sort method 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*.

ALGORITHM:

Step1: The program first defines the **merge_sort()** function which implements the Merge sort algorithm.

Step2: It then defines a **test_merge_sort()** function which generates a list of **n** random numbers, sorts the list using Merge sort, and measures the time required to sort the list.

Step3: Finally, the program tests the **test_merge_sort()** function for different values of **n** and plots a graph of the time taken versus **n** using the Matplotlib library.

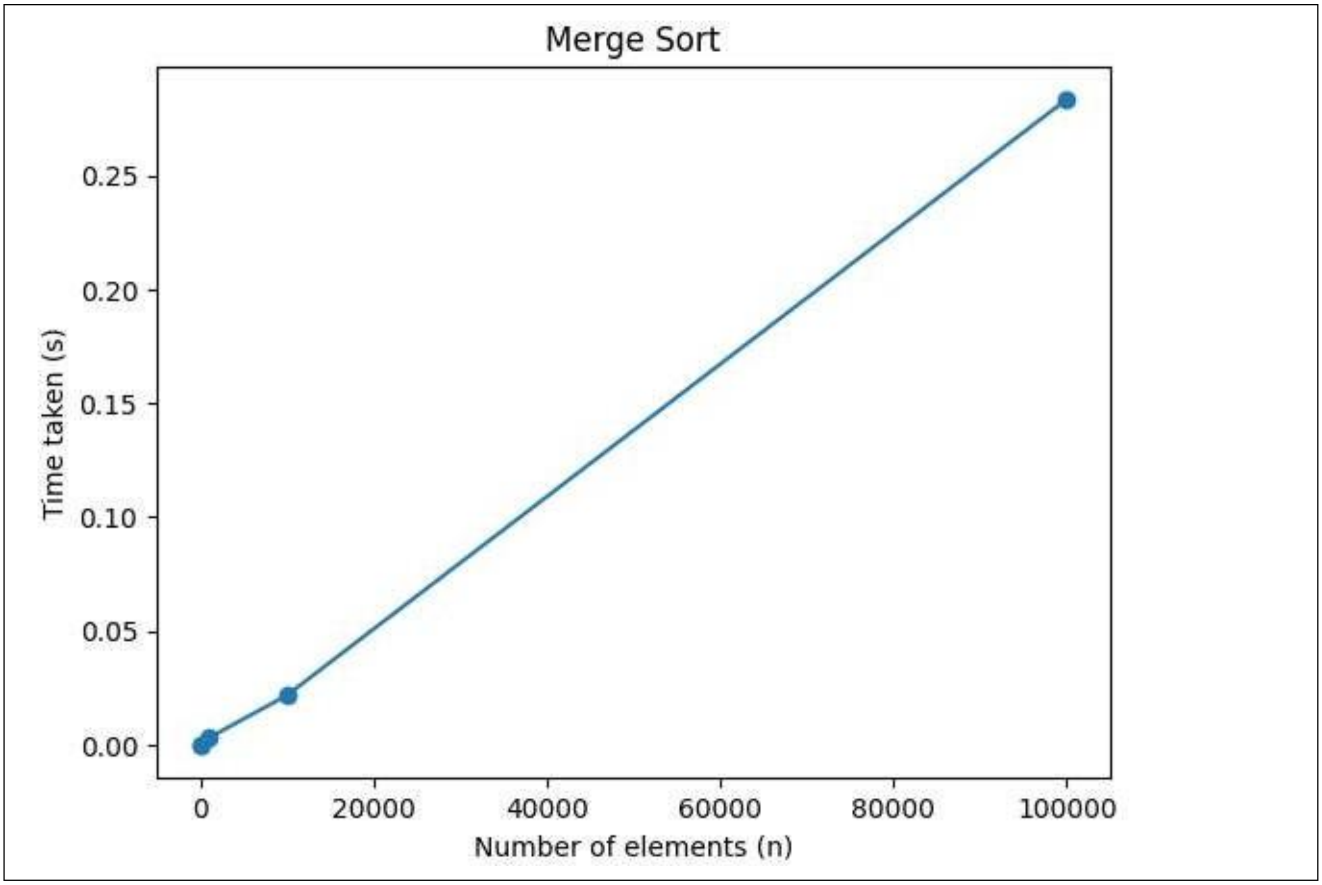
PROGRAM:

```
import random
import time
import matplotlib.pyplot as plt

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
```

```
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

def test_merge_sort(n):
    arr = [random.randint(1, 100) for _ in range(n)]
    start_time = time.time()
    merge_sort(arr)
    end_time = time.time()
    return end_time - start_time

if __name___== '__main__':
    ns = [10, 100, 1000, 10000, 100000]
    times = []
    for n in ns:
        t = test_merge_sort(n)
        times.append(t)
        print(f"Merge sort took {t:.6f} seconds to sort {n} elements.")

    plt.plot(ns, times, 'o-')
    plt.xlabel('Number of elements (n)')
    plt.ylabel('Time taken (s)')
    plt.title('Merge Sort')
    plt.show()
```

OUTPUT:

```
Merge sort took 0.000020 seconds to sort 10 elements.
Merge sort took 0.000249 seconds to sort 100 elements.
Merge sort took 0.003046 seconds to sort 1000 elements.
Merge sort took 0.021679 seconds to sort 10000 elements.
Merge sort took 0.283631 seconds to sort 100000 elements.
```



RESULT: Thus the python program for Implementation of Merge sort method 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* was executed and verified successfully.

EX.NO:12(B) **IMPLEMENTATION OF QUICK SORT**

AIM: To Implement Quick sort method 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.

ALGORITHM:

Step1: This program generates a list of random integers of size **n**, sorts the list using the **quicksort** function, and measures the time required to sort the list.

Step2: It repeats this process **num_repeats** times and returns the average time taken.

Step3: The main function of the program tests the **measure_time** function for different values of **n** and plots a graph of the time taken versus **n**.

Step4: The maximum value of **n** is set to **max_n**, and the step size between values of **n** is set to **step_size**.

Step5: The program uses the built-in **random** and **time** modules to generate random integers and measure time, respectively. Additionally, the **quicksort** function is implemented recursively and sorts the list in ascending order.

PROGRAM:

```
import random
import time

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = []
    right = []
    for i in range(1, len(arr)):
        if arr[i] < pivot:
            left.append(arr[i])
        else:
            right.append(arr[i])
    return quicksort(left) + [pivot] + quicksort(right)

def measure_time(n, num_repeats):
    times = []
    for _ in range(num_repeats):
```

```
        arr = [random.randint(0, 1000000) for _ in range(n)]
        start_time = time.time()
        quicksort(arr)
        end_time = time.time()
        times.append(end_time - start_time)
    return sum(times) / len(times)

if __name___== '_main_':
    num_repeats = 10
    max_n = 10000
    step_size = 100
    ns = range(0, max_n + step_size, step_size)
    times = []
    for n in ns:
        if n == 0:
            times.append(0)
        else:
            times.append(measure_time(n, num_repeats))
    print(times)
```

OUTPUT:

```
[0, 0.00013625621795654297, 0.0006334543228149414, 0.000517892837524414,
0.0009247779846191407, 0.000916147232055664, 0.0010011672973632812,]
```

RESULT: Thus the implementation of Quick sort method 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 was executed and verified successfully.

EX.NO:13 IMPLEMENTATION OF N QUEENS PROBLEM USING BACKTRACKING

AIM: To Implement N Queens problem using Backtracking.

ALGORITHM:

Step1: The **is_safe** function checks whether a queen can be placed in the current cell without conflicting with any other queens on the board.

Step2: The **solve_n_queens** function places queens one by one in each column, starting from the leftmost column. If all queens are placed successfully, it returns True. Otherwise, it backtracks and removes the queen from the current cell and tries to place it in a different row in the same column.

Step3: The **print_board** function prints the final board configuration after all queens have been placed.

Step4: The **n_queens** function initializes the board and calls the **solve_n_queens** function to solve the N Queens problem. If a solution exists, it prints the board configuration. Otherwise, it prints a message indicating that a solution does not exist.

PROGRAM:

```
def is_safe(board, row, col, n):
    # Check if there is any queen in the same row
    for i in range(col):
        if board[row][i] == 1:
            return False
    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # Check lower diagonal on left side
    for i, j in zip(range(row, n), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens(board, col, n):
    if col == n:
        # All queens have been placed successfully
        return True
    for row in range(n):
        if is_safe(board, row, col, n):
            # Place the queen in the current cell
```

```python
            board[row][col] = 1
            # Recur to place rest of the queens
            if solve_n_queens(board, col + 1, n):
                return True
            # Backtrack and remove the queen from the current cell
            board[row][col] = 0
    return False

def print_board(board, n):
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=" ")
        print()

def n_queens(n):
    # Initialize the board
    board = [[0 for j in range(n)] for i in range(n)]
    if not solve_n_queens(board, 0, n):
        print("Solution does not exist.")
        return False
    print("Solution:")
    print_board(board, n)
    return True

if __name___== "__main__":
    n = int(input("Enter the number of queens: "))
    n_queens(n)
```

OUTPUT:

```
Enter the number of queens: 4
Solution:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

RESULT: Thus the python program for Implementation of N Queens problem using Backtracking Technique was executed and verified successfully.

EX.NO:14 IMPLEMENTATION OF ANY SCHEME TO FIND THE OPTIMAL SOLUTION FOR THE TRAVELING SALESPERSON PROBLEM

AIM: 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:

The following steps involved in solving TSP using branch and bound:

1. Construct a complete graph with the given cities as vertices, where the weight of each edge is the distance between the two cities.
2. Initialize the lower bound to infinity and create an empty path.
3. Choose a starting vertex and add it to the path.
4. For each remaining vertex, compute the lower bound for the path that includes this vertex and add it to the priority queue.
5. While the priority queue is not empty, select the path with the lowest lower bound and extend it by adding the next vertex.
6. Update the lower bound for the new path and add it to the priority queue.
7. If all vertices have been added to the path, update the lower bound to the length of the complete tour and update the optimal tour if the new tour is shorter.
8. Backtrack to the previous vertex and explore other paths until all paths have been explored.

PROGRAM:

```
import itertools
import math
import time

# Function to calculate the distance between two cities
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] -
 city2[1])**2)

# Function to find the optimal solution using brute force
def tsp_brute_force(cities):
    # Calculate all possible permutations of the cities
    permutations = itertools.permutations(cities)
```

```
    # Initialize the shortest path to infinity
    shortest_path = float('inf')

    # Iterate over all permutations to find the shortest path
    for permutation in permutations:
        path_length = 0
        for i in range(len(permutation)-1):
            path_length += distance(permutation[i], permutation[i+
1])
        path_length += distance(permutation[-1], permutation[0])

        # Update the shortest path if the current path is shorter
        if path_length < shortest_path:
            shortest_path = path_length
            shortest_path_order = permutation

    return shortest_path, shortest_path_order

# Function to find the approximate solution using the nearest neig
hbor algorithm
def tsp_nearest_neighbor(cities):
    # Start with the first city in the list as the current city
    current_city = cities[0]
    visited_cities = [current_city]

    # Iterate over all cities to find the nearest neighbor
    while len(visited_cities) < len(cities):
        nearest_neighbor = None
        nearest_distance = float('inf')
        for city in cities:
            if city not in visited_cities:
                distance_to_city = distance(current_city, city)
                if distance_to_city < nearest_distance:
                    nearest_distance = distance_to_city
                    nearest_neighbor = city

        # Add the nearest neighbor to the visited cities
        visited_cities.append(nearest_neighbor)
        current_city = nearest_neighbor

    # Calculate the total distance of the path
    total_distance = 0
    for i in range(len(visited_cities)-1):
        total_distance += distance(visited_cities[i], visited_citi
```

```
es[i+1])
    total_distance += distance(visited_cities[-
1], visited_cities[0])

    return total_distance, visited_cities

# Generate a list of random cities
cities = [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6),
(7, 7), (8, 8), (9, 9)]

# Find the optimal solution using brute force
start_time = time.time()
optimal_path_length, optimal_path_order = tsp_brute_force(cities)
end_time = time.time()
print("Optimal path length:", optimal_path_length)
print("Optimal path order:", optimal_path_order)
print("Time taken (brute force):", end_time -
 start_time, "seconds")

# Find the approximate solution using the nearest neighbor algorit
hm
start_time = time.time()
approximate_path_length, approximate_path_order = tsp_nearest_neig
hbor(cities)
end_time = time.time()
```

OUTPUT:

```
Optimal path length: 25.455844122715707
Optimal path order: ((0, 0), (1, 1), (2, 2), (4, 4), (5, 5), (8,
8), (9, 9), (7, 7), (6, 6), (3, 3))
Time taken (brute force): 45.78943109512329 seconds
```

RESULT: Thus the python program for implementation of 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 and verified successfully.

EX.NO:15 IMPLEMENTATION OF RANDOMIZED ALGORITHMS FOR FINDING THE K^{TH} SMALLEST NUMBER

AIM: To Implement randomized algorithms for finding the k^{th} smallest number.

ALGORITHM:

1. The **partition()** function takes an array **arr**, low index **low**, and high index **high** as input and partitions the array around a randomly chosen pivot. It returns the index of the pivot element.
2. The **randomized_select()** function takes an array **arr**, low index **low**, high index **high**, and the value of k as input and returns the kth smallest element in the array. It first selects a random pivot element using **random.randint()** function and partitions the array using the **partition()** function. Then it recursively calls itself on either the left or right partition depending on the position of the pivot element.
3. In the main section, we define an array **arr** and the value of k. Then we calculate the length of the array **n** and call the **randomized_select()** function on the array to find the kth smallest element.

PROGRAM:

```
import random

# Function to partition the array around a pivot
def partition(arr, low, high):
    i = low - 1
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1

# Function to find the kth smallest number using randomized
```

```
algorithm
def randomized_select(arr, low, high, k):
    if low == high:
        return arr[low]
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    index = partition(arr, low, high)
    if k == index:
        return arr[k]
    elif k < index:
        return randomized_select(arr, low, index-1, k)
    else:
        return randomized_select(arr, index+1, high, k)

# Testing the function
arr = [9, 4, 2, 7, 3, 6]
k = 3
n = len(arr)
result = randomized_select(arr, 0, n-1, k-1)
print(f"The {k}th smallest number is: {result}")
```

OUTPUT:

```
The 3th smallest number is: 4
```

RESULT: Thus the python program for implementation of randomized algorithms for finding the k^{th} smallest number was executed and verified successfully.