NPR
COLLEGE OF ENGINEERING & TECHNOLOGY
(AUTONOMOUS)
Approved by AICTE,New Delhi & Affiliated to Anna University,Chennai| Accredited by NAAC with 'A' GRADE
Recognized by UGC under 2 (f) | ISO 9001:2015 Certified | Web: www.nprcolleges.org | E-Mail: nprcetprincipal@nprcolleges.org
NPR Nagar, Natham - 624 401, Dindigul Dist, Tamil Nadu. Ph: 04544 - 246500, 501, 502.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3401- ALGORITHMS LABORATORY

Semester IV

Lab Manual

Regulation 2021

TABLE OF CONTENTS

NPR COLLEGE OF ENGINEERING & TECHNOLOGY, NATHAM

VISION

- To develop students with intellectual curiosity and technical expertise to meet the global needs.

MISSION

- To achieve academic excellence by offering quality technical education using best teaching techniques.
- To improve Industry – Institute interactions and expose industrial atmosphere.
- To develop interpersonal skills along with value based education in a dynamic learning environment.
- To explore solutions for real time problems in the society.

NPR COLLEGE OF ENGINEERING & TECHNOLOGY, NATHAM

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

- To produce globally competent technical professionals for digitized society.

MISSION

- To establish conducive academic environment by imparting quality education and value added training.
- To encourage students to develop innovative projects to optimally resolve the challenging social problems.

PROGRAM EDUCATIONAL OBJECTIVES

Graduates of Computer Science and Engineering Program will be able to:

- Develop into the most knowledgeable professional to pursue higher education and Research or have a successful carrier in industries.
- Successfully carry forward domain knowledge in computing and allied areas to solve complex and real world engineering problems.
- Meet the technological revolution they are continuously upgraded with the technical knowledge.
- Serve the humanity with social responsibility combined with ethics

CS3401 ALGORITHMS LABORATORY L T P C

3 0 2 4

OBJECTIVES:

- To understand and apply the algorithm analysis techniques for searching and sorting algorithms
- To critically analyze the efficiency of graph algorithms
- To understand different algorithm design techniques
- To solve programming problems using a state space tree
- To understand the concepts behind NP-Completeness, Approximation algorithms and randomized algorithms.

LIST OF EXPERIMENTS:

A. 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
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.
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.
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.

B. Graph Algorithms

1. Develop a program to implement graph traversal using Breadth First Search
2. Develop a program to implement graph traversal using Depth First Search
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.
4. Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.
5. Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem.
6. Compute the transitive closure of a given directed graph using Warshall's algorithm.

C. 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.
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.

D. State Space Search Algorithms

1. Implement N Queens problem using Backtracking.

E. 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.

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

The programs can be implemented in C/C++/JAVA/ Python.

TOTAL:30 PERIODS

OUTCOMES:

At the end of this course, the students will be able to:

CO1: Analyze the efficiency of algorithms using various frameworks

CO2: Apply graph algorithms to solve problems and analyze their efficiency.

CO3: Make use of algorithm design techniques like divide and conquer, dynamic programming and greedy techniques to solve problems

CO4: Use the state space tree method for solving problems.

CO5: Solve problems using approximation algorithms and randomized algorithms

CS3401 **ALGORITHMS LABORATORY**

Course Outcomes

After completion of the course, Students are able to learn the listed Course Outcomes.

Cos	Course Code	Course Outcomes	Knowledge Level
CO1	C214.1	Analyze the efficiency of algorithms using various frameworks	K4
CO2	C214.2	Apply graph algorithms to solve problems and analyze their efficiency.	K3
CO3	C214.3	Make use of algorithm design techniques like divide and conquer, dynamic programming and greedy techniques to solve problems	K3
CO4	C214.4	Use the state space tree method for solving problems.	K3
CO5	C214.5	Solve problems using approximation algorithms and randomized algorithms	K3

List of Experiments with COs, POs and PSOs

Exp.No.	Name of the Experiment	COs	POs	PSOs
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	CO1	PO1,2,3	PSO1,2
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.	CO1	PO1,2,3	PSO1,2
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.	CO1	PO1,2,3	PSO1,2
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	CO1	PO1,2,3	PSO1,2

	elements in the list to be sorted and plot a graph of the time taken versus n.			
5.	Develop a program to implement graph traversal using Breadth First Search	CO2	PO1,2,3	PSO1,2
6.	Develop a program to implement graph traversal using Depth First Search	CO2	PO1,2,3	PSO1,2
7.	From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra's algorithm.	CO2	PO1,2,3	PSO1,2
8.	Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.	CO2	PO1,2,3	PSO1,2
9.	Implement Floyd's algorithm for the All-Pairs-Shortest-Paths problem.	CO2	PO1,2,3	PSO1,2
10.	Compute the transitive closure of a given directed graph using Warshall's algorithm.	CO2	PO1,2,3	PSO1,2
11.	Develop a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.	CO3	PO1,2,3	PSO1,2
12.	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.	CO3	PO1,2,3	PSO1,2
13.	Implement N Queens problem using Backtracking	CO4	PO1,2,3	PSO1,2
14.	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.	CO5	PO1,2,3	PSO1,2
15.	Implement randomized algorithms for finding the kth smallest number.	CO5	PO1,2,3	PSO1,2

Program Outcomes

1. Engineering Knowledge
2. Problem Analysis
3. Design/Development of Solutions
4. Conduct Investigations of Complex Problems
5. Modern Tool Usage
6. The Engineer and Society
7. Environment and Sustainability
8. Ethics
9. Individual and Team Work
10. Communication
11. Project Management and Finance
12. Life-long Learning

Program Specific Outcomes

At the end of the program students will be able to

- Deal with real time problems by understanding the evolutionary changes in computing, applying standard practices and strategies in software project development using open-ended programmingenvironments.
- Employ modern computer languages, environments and platforms in creating innovative career pathsby inculcating moral values and ethics.
- Achieve additional expertise through add-on and certificate programs.

Ex.No: A1 Date:	SEARCHING ALGORITHMS

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 searching from the first element of the list

Step 2: Compare the target element with the current element

Step 3: If they match, return the index of the current element

Step 4: If not, move to the next element and repeat steps 2-3 until the end of the list is reached

Step 5: If the end of the list is reached and the target element is not found, return -1

To determine the time taken for linear search for different values of n,

Step 1: Generate a list of n elements (randomly or in a specific pattern)

Step 2: Choose an element to search for (either present or not present in the list)

Step 3: Start a timer and perform linear search on the list for the chosen element

Step 4: Stop the timer when the element is found or after the search is complete

Step 5: Repeat steps 2-4 multiple times for each value of n and calculate the average time taken

Step 6: Plot a graph of the average time taken versus n

Program:

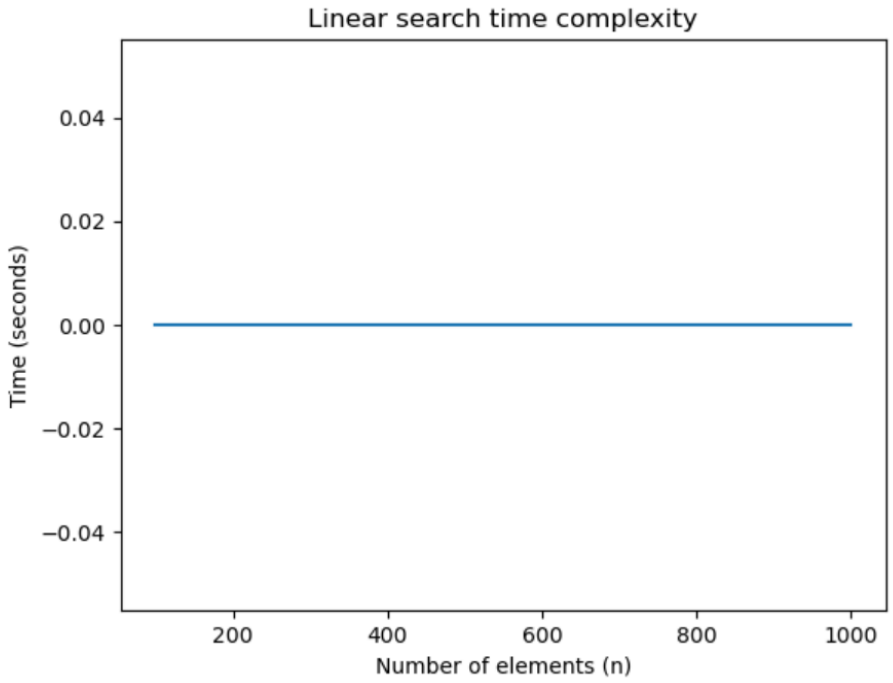
```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

#To determine the time required to search for an element, we can use the time module in Python to measure the elapsed time:#

```
import time
import random
import matplotlib.pyplot as plt
def measure_time(n):
    arr = [random.randint(0, 100) for i in range(n)]
    x = arr[random.randint(0, n-1)]
    start_time = time.time()
    linear_search(arr, x)
    end_time = time.time()
    return end_time - start_time
```

```
ns = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
times = [measure_time(n) for n in ns]
plt.plot(ns, times)
plt.xlabel('Number of elements (n)')
plt.ylabel('Time (seconds)')
plt.title('Linear search time complexity')
plt.show()
```

Output:



Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: A2 Date:	SEARCHING ALGORITHMS

Aim:

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: Define a function binary_search(arr, low, high, x) that takes the array, lower index, upper index and the element to be searched as inputs.

Step 2: Check if the upper index is greater than or equal to the lower index. If it is not, return -1 (element not found).

Step 3: Calculate the middle index using the formula mid = (low + high) / 2.

Step 4: Compare the middle element of the array with the element to be searched. If they are equal, return the index of the middle element.

Step 5: If the middle element is greater than the element to be searched, recursively call the binary_search function with the lower half of the array: binary_search(arr, low, mid-1, x).

Step 6: If the middle element is less than the element to be searched, recursively call the binary_search function with the upper half of the array: binary_search(arr, mid+1, high, x).

Step 7: If the element is not found in the array, return -1 (element not found).

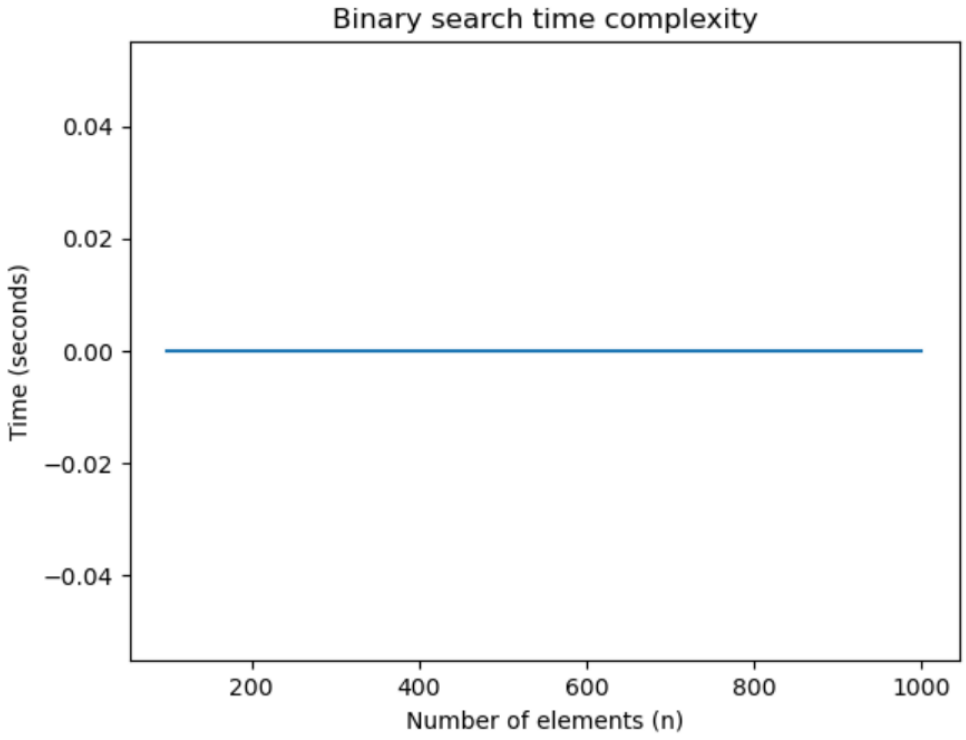
Time Complexity of Binary Search: O(log n)

Program:

```
def binary_search_recursive(arr, l, r, x):
    if r >= l:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search_recursive(arr, l, mid-1, x)
        else:
            return binary_search_recursive(arr, mid+1, r, x)
    else:
        return -1
#To determine the time required to search for an element, we can use the time module in Python to measure the elapsed time:
import time
import random
import matplotlib.pyplot as plt
def measure_time(n):
    arr = sorted([random.randint(0, 100) for i in range(n)])
    x = arr[random.randint(0, n-1)]
```

```
    start_time = time.time()
    binary_search_recursive(arr, 0, n-1, x)
    end_time = time.time()
    return end_time - start_time
ns = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]
times = [measure_time(n) for n in ns]
plt.plot(ns, times)
plt.xlabel('Number of elements (n)')
plt.ylabel('Time (seconds)')
plt.title('Binary search time complexity')
plt.show()
```

Output:



Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: A3 Date:	SEARCHING ALGORITHMS

Aim:

To pattern search, 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.

Algorithm:

Step 1: Initialize two integers i and j to 0.

Step 2: While i is less than or equal to n-m, do the following:

Step 3: Compare the characters of pat and txt starting from index j for m times.

Step 4: If they all match, print the index i.

Step 5: Increment both i and j.

Step 6: Return.

Program:

```
#include <iostream>
#include <string.h>
using namespace std;
void search(char pat[], char txt[])
{
   int m = strlen(pat);
   int n = strlen(txt);
   for (int i = 0; i <= n - m; i++)
   {
      int j;
      for (j = 0; j < m; j++)
      {
         if (txt[i + j] != pat[j])
            break;
      }
      if (j == m)
      {
         cout << "Pattern found at index " << i << endl;
      }
   }
}
int main()
{
   char txt[] = "AABAACAADAABAAABAA";
```

```
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: A4 Date:	SORTING ALGORITHMS

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 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:

Insertion Sort Algorithm:

Step 1: For each index i from 1 to n-1, do the following:

Step 2: Let temp = arr[i]

Step 3: Let j = i-1

Step 4: While j >= 0 and arr[j] > temp, do the following:

Step 5: arr[j+1] = arr[j]

Step 6: Decrement j by 1

Step 7: Let arr[j+1] = temp

Heap Sort Algorithm:

Step 1: Build a max heap out of the given array.

Step 2: For each index i from n-1 down to 1, do the following:

Step 3: Swap arr[0] with arr[i]

Step 4: Heapify the array from index 0 to i-1.

Program:

```
import random
import time
import matplotlib.pyplot as plt
def insertion_sort(arr):
for i in range(1, len(arr)):
temp = arr[i]
j = i - 1
while j >= 0 and arr[j] > temp:
arr[j+1] = arr[j]
j -= 1
arr[j+1] = temp
def heap_sort(arr):
def heapify(arr, n, i):
largest = i
l = 2*i + 1
r = 2*i + 2
```

```
if l < n and arr[l] > arr[largest]:
largest = l
if r < n and arr[r] > arr[largest]:
largest = r
if largest != i:
arr[i], arr[largest] = arr[largest], arr[i]
heapify(arr, n, largest)
n = len(arr)
for i in range(n//2 - 1, -1, -1):
heapify(arr, n, i)
for i in range(n-1, 0, -1):
arr[0], arr[i] = arr[i], arr[0]
heapify(arr, i, 0)
# Testing the algorithms for different values of n
n_list = [100, 1000, 5000, 10000, 15000, 20000]
insertion_sort_times = []
heap_sort_times = []
for n in n_list:
arr = [random.randint(0, n) for i in range(n)]
start_time = time.time()
insertion_sort(arr)
end_time = time.time()
insertion_sort_times.append(end_time - start_time)
start_time = time.time()
heap_sort(arr)
end_time = time.time()
heap_sort_times.append(end_time - start_time)
# Plotting the graph
plt.plot(n_list, insertion_sort_times, label="Insertion Sort")
plt.plot(n_list, heap_sort_times, label="Heap Sort")
plt.xlabel("")
```

Output:

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: B1 Date:	GRAPH ALGORITHMS

Aim:

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

Algorithm:

Step 1: Create a Graph class that holds the nodes and edges of the graph.

Step 2: Define a Node class that holds the characteristics of each node such as their value, list of neighbors and visited status.

Step 3: Implement a Breadth First Search (BFS) traversal function that takes in a starting node and processes the rest of the graph by visiting its neighbors first before visiting their neighbor's neighbors.

Step 4: Initialize an empty queue and add the starting node to it.

Step 5: While the queue is not empty, remove the first node from the queue and mark it as visited.

Step 6: Visit all unvisited neighbors of the removed node and add them to the queue.

Step 7: Continue until all nodes have been visited.

Step 8: Return the order in which nodes were visited.

Program:

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self, s):
        visited = [False] * (len(self.graph))
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            s = queue.pop(0)
            print (s, end = " ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
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
```

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: B2 Date:	GRAPH ALGORITHMS

Aim:

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

Algorithm:

Step 1: Create a class to represent a graph with a list of vertices.

Step 2: Define a method to add vertices to the graph.

Step 3: Define a method to add edges between vertices in the graph.

Step 4: Define a method to perform Depth First Search on the graph.

Step 5: Initialize a stack and add the starting vertex to it.

Step 6: Loop until the stack is empty:

Step 7: Pop a vertex from the stack.

Step 8: If the vertex has not been visited, mark it as visited and process it.

Step 9: For each adjacent vertex to the current vertex, if it has not been visited, add it to the stack.

Step 10: Once all vertices have been processed, terminate the algorithm.

Program:

```
class Graph:
def __init__(self):
self.vertices = {}
def add_vertex(self, vertex):
self.vertices[vertex] = []
def add_edge(self, vertex1, vertex2):
self.vertices[vertex1].append(vertex2)
self.vertices[vertex2].append(vertex1)
def DFS(self, start_vertex):
visited = set()
stack = [start_vertex]
while stack:
current_vertex = stack.pop()
if current_vertex not in visited:
visited.add(current_vertex)
print(current_vertex)
for adjacent_vertex in self.vertices[current_vertex]:
if adjacent_vertex not in visited:
stack.append(adjacent_vertex)
```

```
g = Graph()
g.add_vertex(0)
g.add_vertex(1)
g.add_vertex(2)
g.add_vertex(3)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.DFS(0)
```

Output:

0

2

3

1

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: B3 Date:	GRAPH ALGORITHMS

Aim:

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

Algorithm:

Step 1: Define the graph and the source vertex

Step 2: Initialize data structures

Step 3: Loop through unvisited vertices

Step 4: Trace the shortest path to each vertex

Step 5: Output the results

Program:

```
import sys
import heapq
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[] for i in range(vertices)]
    def add_edge(self, u, v, w):
        self.graph[u].append((v, w))
    def shortest_path(self, start):
        dist = [sys.maxsize] * self.V
        dist[start] = 0
        heap = [(0, start)]
        while heap:
            (d, u) = heapq.heappop(heap)
            if d > dist[u]:
                continue
            for v, w in self.graph[u]:
                if dist[v] > dist[u] + w:
                    dist[v] = dist[u] + w
                    heapq.heappush(heap, (dist[v], v))
        return dist

g = Graph(9)
g.add_edge(0, 1, 4)
g.add_edge(0, 7, 8)
```

```
g.add_edge(1, 2, 8)
g.add_edge(1, 7, 11)
g.add_edge(2, 3, 7)
g.add_edge(2, 8, 2)
g.add_edge(2, 5, 4)
g.add_edge(3, 4, 9)
g.add_edge(3, 5, 14)
g.add_edge(4, 5, 10)
g.add_edge(5, 6, 2)
g.add_edge(6, 7, 1)
g.add_edge(6, 8, 6)
g.add_edge(7, 8, 7)
start = 0
print("The shortest distances from vertex", start)
distances = g.shortest_path(start)
for i in range(len(distances)):
   print("to vertex", i, "is", distances[i])
```

Output:

```
The shortest distances from vertex 0
to vertex 0 is 0
to vertex 1 is 4
to vertex 2 is 12
to vertex 3 is 19
to vertex 4 is 28
to vertex 5 is 16
to vertex 6 is 18
to vertex 7 is 8
to vertex 8 is 14
```

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: B4 Date:	GRAPH ALGORITHMS

Aim:

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

Algorithm:

Step 1: Initialize an empty set of vertices, called MST (minimum spanning tree), and a set of all vertices in the graph, called V.

Step 2: Choose any vertex, v0, from V and add it to MST.

Step 3: Create a priority queue, Q, of all edges adjacent to v0 with weights as their keys.

Step 4: While Q is not empty:

- Remove the edge, e, with the minimum weight from Q.

- If e connects two vertices in MST, discard it.

- Otherwise, add the connecting vertex, v, to MST and add all edges adjacent to v with endpoints not in MST to Q.

Step 5: Repeat step 4 until MST contains all vertices in V.

Step 6: Return MST as the minimum cost spanning tree.

Program:

```
from queue import PriorityQueue
def prim(graph, start):
    mst = []
    visited = set()
    pq = PriorityQueue()
    pq.put((0, start))
    while not pq.empty():
        weight, node = pq.get()
        if node in visited:
            continue
        visited.add(node)
        mst.append((node, weight))
        for neighbor, w in graph[node].items():
            if neighbor not in visited:
                pq.put((w, neighbor))
    return mst
graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'A': 2, 'C': 1, 'D': 3},
    'C': {'A': 3, 'B': 1, 'D': 2},
    'D': {'B': 3, 'C': 2}
```

```
}
print(prim(graph, 'A'))
```

Output:

```
[('A', 0), ('B', 2), ('C', 1), ('D', 2)]
```

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: B5 Date:	GRAPH ALGORITHMS

Aim:

To implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem.

Algorithm:

Step 1. Create a 2D array D of size n x n, where n is the number of vertices in the graph.

Step 2. Set all the diagonal elements of D to 0 and all the other elements to infinity.

Step 3. For each edge (u,v) with weight w in the graph, set D[u][v] = w.

Step 4. The final 2D array D contains the shortest paths between all pairs of vertices.

Step 5. If there are negative-weight cycles in the graph, the above algorithm may not terminate or may return incorrect results. It is necessary to check for the existence of negative-weight cycles using Bellman-Ford or other algorithms before applying Floyd's algorithm.

Program:

```
def floyd_warshall(graph):
    n = len(graph)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])
    return graph
graph = [
    [0, float('inf'), -2, float('inf')],
    [4, 0, 3, float('inf')],
    [float('inf'), float('inf'), 0, 2],
    [float('inf'), -1, float('inf'), 0]
]
print(floyd_warshall(graph))
```

Output:

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

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: B6 Date:	GRAPH ALGORITHMS

Aim:

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

Algorithm:

Step 1. Create a matrix A with the adjacency matrix of the given graph.

Step 2. For each vertex i, set A[i][i] to 1. This ensures that all vertices are reachable from themselves.

Step 3. For each pair of vertices i and j, if there is an edge from i to j, set A[i][j] to 1. Otherwise, set it to 0.

Step 4. For each pair of vertices i and j, if there is a path from i to j (including direct edges), set A[i][j] to 1.

Step 5. Repeat step 4 until no more changes are made to A.

Step 6. The resulting matrix A is the transitive closure of the given graph.

Program:

```
def transitive_closure(graph):
    """
    Computes the transitive closure of a given directed graph using Warshall's algorithm.
    Parameters:
    graph (list of lists): The input graph represented as an adjacency matrix
    Returns:
    list of lists: The transitive closure of the input graph.
    """
    n = len(graph)
    reach = [[False for j in range(n)] for i in range(n)]
    for i in range(n):
        for j in range(n):
            reach[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])
    return reach
graph = [[1, 1, 0, 1], [0, 1, 1, 0], [0, 0, 1, 1], [0, 0, 0, 1]]
print(transitive_closure(graph))
```

Output:

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

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: C1 Date:	ALGORITHM DESIGN TECHNIQUES

Aim:

To develop a 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. Divide the list into two halves.

Step 2. Recursively find the maximum and minimum values in each half.

Step 3. Compare the maximum and minimum values of each half to find the overall maximum and minimum values of the list.

Step 4. Return the maximum and minimum values found in step 3.

Program:

```
def divide_and_conquer_min_max(arr, low, high):
    if low == high:
        return arr[low], arr[high]
    if low + 1 == high:
        return min(arr[low], arr[high]), max(arr[low], arr[high])
    mid = (low + high) // 2
    min_left, max_left = divide_and_conquer_min_max(arr, low, mid)
    min_right, max_right = divide_and_conquer_min_max(arr, mid+1, high)
    return min(min_left, min_right), max(max_left, max_right)
def get_min_max(arr):
    return divide_and_conquer_min_max(arr, 0, len(arr)-1)
arr = [3, 5, 2, 1, 9, 7]
min_num, max_num = get_min_max(arr)
print("Minimum number:", min_num)
print("Maximum number:", max_num)
```

Output:

Minimum number: 1
Maximum number: 9

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: C2 Date:	ALGORITHM DESIGN TECHNIQUES

Aim:

To 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.

Algorithm:

1. Merge Sort Algorithm

a. Divide the unsorted array into two halves, until each half has one or zero elements.

b. Recursively sort each half of the array.

c. Merge the sorted halves to produce the final sorted array.

2. Quick Sort Algorithm

a. Choose a pivot element from the array.

b. Partition the array into two sub-arrays, such that all elements less than the pivot are on the left and all elements greater than the pivot are on the right.

c. Recursively sort the left and right sub-arrays.

a. For Merge sort, the time complexity is O(nlogn), as the array is divided into halves recursively and merged back together.

b. For Quick sort, the time complexity is O(n^2) in the worst case, but on average it is O(nlogn).

a. Generate random arrays of integers of varying sizes.

b. Sort the array using both merge sort and quick sort.

c. Record the time taken by both algorithms for each array size.

d. Plot a graph with the x-axis representing the array size and the y-axis representing the time taken.

Program:

Python Code for Merge Sort:

```
import time
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
arr = [10, 7, 8, 9, 1, 5]
t0 = time.time()
merge_sort(arr)
t1 = time.time()
print("Sorted array:", arr)
print(f"Time taken: {t1-t0} seconds")
```

Python Code for QuickSort:

```
import time
def partition(arr, low, high):
i = (low-1)
pivot = arr[high]
for j in range(low, high):
if arr[j] <= pivot:
i = i+1
arr[i], arr[j] = arr[j], arr[i]
arr[i+1], arr[high] = arr[high], arr[i+1]
return (i+1)
def quick_sort(arr, low, high):
if low < high:
pi = partition(arr, low, high)
```

```
quick_sort(arr, low, pi-1)
quick_sort(arr, pi+1, high)
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
t0 = time.time()
quick_sort(arr, 0, n-1)
t1 = time.time()
print("Sorted array:", arr)
print(f"Time taken: {t1-t0} seconds")
```

Now we can test both algorithms based on different input sizes:

```
import random
import matplotlib.pyplot as plt
def make_plot(n_list, time_list, title, xlabel, ylabel):
plt.plot(n_list, time_list)
plt.title(title)
plt.xlabel(xlabel)
plt.ylabel(ylabel)
plt.show()
def test_sort(n):
arr_m = [random.randint(0, 100000) for _ in range(n)]
arr_q = arr_m.copy()
t0 = time.time()
merge_sort(arr_m)
t1 = time.time()
time_m = t1 - t0
t0 = time.time()
quick_sort(arr_q, 0, n-1)
t1 = time.time()
time_q = t1 - t0
return time_m, time_q
n_list = [10**x for x in range(1, 6)]
time_m_list = []
time_q_list = []
for n in n_list:
time_m, time_q = test_sort(n)
time_m_list.append(time_m)
time_q_list.append(time_q)
```

```
make_plot(n_list, time_m_list, "Merge Sort", "n", "Time (s)")
make_plot(n_list, time_q_list, "Quick Sort", "n", "Time (s)")
```

Output:

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: D1 Date:	STATE SPACE SEARCH ALGORITHMS

Aim:

To implement the N Queens problem using Backtracking

Algorithm:

Step 1. Initialize an empty chessboard of size n x n.

Step 2. Start by placing the first queen in the first row and column.

Step 3. While there is still an empty cell on the board, repeat steps 4-8.

Step 4. Choose an empty cell in the next row (i.e., the row immediately below the last queen), starting from the leftmost column.

Step 5. Check if it is safe to place a queen in that cell. A cell is safe if no other queens can attack it. To check this, we need to examine three things:

a. Check if there is any queen already placed in the same column as the selected cell.

b. Check if there is any queen already placed in the diagonal line going from top-left to bottom-right of the selected cell.

c. Check if there is any queen already placed in the diagonal line going from the top-right to the bottom-left of the selected cell.

Step 6. If the chosen cell is safe, place a queen in that cell.

Step 7. If all rows have been filled with queens, then we have found a solution. Print the solution and continue searching for more solutions.

Step 8. If the chosen cell is not safe or there are no safe cells left in the current row, move back to the previous row and try placing the queen in the next available cell. If there are no more available cells, move back to another row and repeat the process.

Step 9. If we have tried all possibilities and still cannot find a solution, backtrack further until all possibilities have been exhausted.

Program:

```
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()
def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
```

```
            return False
    return True
def solveNQUtil(board, col):
    if col == N:
        return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False
def solveNQ():
    board = [[0 for x in range(N)] for y in range(N)]
    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False
    printSolution(board)
    return True
N = 4
solveNQ()
```

Output:

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

True

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: E1 Date:	APPROXIMATION ALGORITHMS RANDOMIZED ALGORITHMS

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:

1. Implementing an optimal solution for TSP:

Step 1: Input the number of cities (n) and a distance matrix of size n x n (d[i][j] represents the distance between city i and j).

Step 2: Create a list of all possible permutations of cities, excluding the starting city (let's assume it is always city 0).

Step 3: For each permutation, calculate its total distance by summing up the distances between consecutive cities in the permutation, and add the distance from the last city back to the starting city.

Step 4: Choose the permutation with the smallest total distance as the solution.

Step 5: Output the solution.

2. **Solving the same problem using an approximation algorithm:**

Step 1: Input the number of cities (n) and a distance matrix of size n x n (d[i][j] represents the distance between city i and j).

Step 2: Create an empty solution list and add the starting city (city 0) to it.

Step 3: While the solution list does not contain all cities:

- Calculate the distance from the current city (last city in the solution list) to all remaining cities.

- Choose the city with the smallest distance and add it to the solution list.

Step 4: Add the distance from the last city in the solution list back to the starting city (city 0) to complete the tour.

Step 5: Output the solution.

Program:

```
import itertools
import networkx as nx
def tsp_exact(G):
    """Find exact solution to TSP using brute-force method."""
    nodes = list(G.nodes())
    best_tour = None
    best_length = float("inf")
    for tour in itertools.permutations(nodes):
        length = sum(G[tour[i]][tour[i + 1]]["weight"] for i in range(len(nodes) - 1)) +
G[tour[-1]][tour[0]]["weight"]
        if length < best_length:
```

```
            best_length = length
            best_tour = tour
    return best_tour, best_length
def tsp_approx(G):
    """Find approximation solution to TSP using Christofides algorithm."""
    nodes = list(G.nodes())
    MST = nx.minimum_spanning_tree(G)
    odd_nodes = [node for node in nodes if len(MST[node]) % 2 == 1]
    matchings = nx.bipartite.maximum_matching(MST, odd_nodes)
    T = nx.Graph()
    T.add_edges_from(MST.edges())
    T.add_edges_from((node, match) for node, match in matchings.items())
    Euler_tour = nx.eulerian_circuit(T, source=odd_nodes[0])
    tour = list(itertools.chain(*(path[:-1] for path in Euler_tour)))
    tour = tour + [tour[0]]
    length = sum(G[tour[i]][tour[i + 1]]["weight"] for i in range(len(nodes)))
    return tour, length
```

Output:

Result:

Thus the above program was successfully executed and the output was obtained

Ex.No: E2 Date:	APPROXIMATION ALGORITHMS RANDOMIZED ALGORITHMS

Aim:

To implement randomized algorithms for finding the kth smallest number.

Algorithm:

Step 1: Define a function that takes an array of integers as input and returns the kth smallest number in the array.

Step 2: Determine the length of the array, n.

Step 3: If k is greater than n or less than 1, return an error message indicating that the kth smallest number does not exist.

Step 4: If k is equal to 1, return the smallest element in the array.

Step 5: If k is equal to n, return the largest element in the array.

Step 6: Randomly select a pivot element from the array.

Step 7: Partition the array into two subarrays: one containing all elements less than the pivot element and one containing all elements greater than the pivot element.

Step 8: Determine the number of elements in the subarray containing elements less than the pivot element, m.

Step 9: If k is equal to m + 1, return the pivot element.

Step 10: If k is less than m + 1, recursively call the function on the subarray containing elements less than the pivot element and the kth smallest number.

Step 11: If k is greater than m + 1, recursively call the function on the subarray containing elements greater than the pivot element and the (k - m - 1)th smallest number.

Step 12: Repeat steps 6-11 until the kth smallest element is found.

Program:

```
import random
def quick_select(array, k, pivot_fn):
    """Return the kth smallest element in the array using pivot_fn."""
    if len(array) == 1:
        return array[0]
    pivot = pivot_fn(array)
    lows = [el for el in array if el < pivot]
    highs = [el for el in array if el > pivot]
    pivots = [el for el in array if el == pivot]
    if k < len(lows):
        return quick_select(lows, k, pivot_fn)
    elif k < len(lows) + len(pivots):
        return pivots[0]
    else:
        return quick_select(highs, k - len(lows) - len(pivots), pivot_fn)
```

```
def random_pivot(array):
    """Return a random pivot element."""
    return random.choice(array)
```

Output:

Result:

Thus the above program was successfully executed and the output was obtained