ANNA UNIVERSITY, CHENNAI
NON - AUTONOMOUS AFFILIATED COLLEGES
REGULATIONS 2021
CHOICE BASED CREDIT SYSTEM

CS3401 ALGORITHMS

L	T	P	C
3	0	2	4

COURSE OBJECTIVES

1. To understand and apply the algorithm analysis techniques on searching and sorting Algorithms.
2. To critically analyze the efficiency of graph algorithms.
3. To understand different algorithm design techniques.
4. To solve programming problems using state space tree.
5. To understand the concepts behind NP Completeness, Approximation algorithms and randomized algorithms.

PRACTICAL EXERCISES: 30 PERIODS

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.

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.

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.

State Space Search Algorithms

1. Implement N Queens problem using Backtracking.

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: 75 PERIODS

COURSE 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.

CHENNAI INSTITUTE OF TECHNOLOGY AND APPLIED RESEARCH

INDEX

S.No	Date	Name of the Experiment	CO's Mapped	PO's & PSO's Mapped	Signature
Searching &Sorting Algorithms					
1.		Implementation of Linear Search			
2.		Implementation of Binary Search			
3.		Searching of a given Text			
4.		Implementations of Insertion sort and Heap sort			
Graph Algorithms					
5.		Implement graph traversal using Breadth First Search			
6.		Implement graph traversal using Depth First Search			
7.		Find the shortest paths to other vertices using Dijkstra's algorithm.			
8.		Implementation of Prim's algorithm.			
9.		Implementation of Floyd's algorithm			
10.		Implementation of Warshall's algorithm			
Algorithm Design Techniques					
11.		Find out the maximum and minimum numbers using the divide and conquer technique.			
12.		Merge sort and Quick sort methods			
State Space Search Algorithms					
13.		N Queen's Problem			
Approximation Algorithms Randomized Algorithms					
14.		Traveling Salesperson problem			
15.		Randomized Algorithm			

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 a linear search algorithm in Python, measure the time taken for searching an element in lists of varying sizes, and plot a graph of time taken versus the size of the list.

Algorithm:

1. **Input**: A list of integers, the size of the list (nnn), and the target element (xxx).
2. **Process**:
   - Traverse each element in the list sequentially.
   - If the element matches the target, return its index.
   - If no match is found after the traversal, return -1.
3. **Output**: Index of the target element if found; otherwise, -1.

Program:

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

# Linear search implementation
def linear_search(arr, target):
    for index, element in enumerate(arr):
        if element == target:
            return index
    return -1

# Experiment function
def measure_time(n_values):
    times = []
    for n in n_values:
        arr = [random.randint(1, 10000) for _ in range(n)]  # Generate random list
        target = random.choice(arr)  # Choose a random target from the list

        # Measure time for linear search
        start_time = time.time()
        linear_search(arr, target)
        end_time = time.time()

        times.append(end_time - start_time)  # Append time taken
    return times

# Define list sizes
n_values = [1000, 2000, 5000, 10000, 20000, 50000, 100000]
times = measure_time(n_values)

# Plotting the graph
```

```
    plt.plot(n_values, times, marker='o', color='b')
    plt.title("Linear Search: Time Taken vs List Size")
    plt.xlabel("Number of Elements (n)")
    plt.ylabel("Time Taken (seconds)")
    plt.grid(True)
    plt.show()
```

Output:

Result:

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 implement a recursive **Binary Search** algorithm in Python, measure the time required to search for an element in a list, and plot a graph of the time taken versus the number of elements (nnn) in the list. We will test for different values of nnn and analyze the performance.

Algorithm:

1. **Input**: A sorted list of integers arr[]arr[]arr[], the size of the list nnn, and the target element xxx.
2. **Process**:
    - Find the middle element of the list.
    - If the middle element is the target, return its index.
    - If the target is smaller than the middle element, search the left half of the list.
    - If the target is larger than the middle element, search the right half of the list.
3. **Output**: Index of the target element if found; otherwise, return -1.

Program:

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

# Recursive binary search function
def binary_search(arr, target, low, high):
    if high >= low:
        mid = (low + high) // 2

        # Check if the target is at mid
        if arr[mid] == target:
            return mid

        # If target is smaller, ignore the right half
        elif arr[mid] > target:
            return binary_search(arr, target, low, mid - 1)

        # Else, ignore the left half
        else:
            return binary_search(arr, target, mid + 1, high)

    # Target is not present in the list
    return -1

# Function to measure time for binary search
def measure_time(n_values):
    times = []
    for n in n_values:
        # Generate a sorted list
        arr = sorted([random.randint(1, 10000) for _ in range(n)])
        target = random.choice(arr)  # Choose a random target from the list

        # Measure time for binary search
```

```
        start_time = time.time()
        binary_search(arr, target, 0, len(arr) - 1)
        end_time = time.time()

        times.append(end_time - start_time)  # Append time taken
    return times

# Define list sizes
n_values = [1000, 2000, 5000, 10000, 20000, 50000, 100000]
times = measure_time(n_values)

# Plotting the graph
plt.plot(n_values, times, marker='o', color='b')
plt.title("Recursive Binary Search: Time Taken vs List Size")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.grid(True)
plt.show()
```

Output

Result:

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 find all occurrences of a pattern pat[] in a text txt[], we can implement a **Naive String Matching** algorithm. This algorithm checks all possible positions in the text where the pattern could match, and compares the substring in the text with the pattern.

Algorithm

1. **Input**: The pattern pat[] and the text txt[].
2. **Process**:
    - Iterate through each possible starting index in the text (from 0 to n - m).
    - At each position, compare the substring of the text with the pattern.
    - If they match, print the index of the occurrence.
3. **Output**: The starting indices of all occurrences of the pattern pat[] in the text txt[].

Naive String Matching Approach

- This algorithm runs in O((n−m+1)×m)O((n-m+1) \times m)O((n−m+1)×m), where nnn is the length of the text and mmm is the length of the pattern.

Python Code Implementation

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

    # Loop through the text
    for i in range(n - m + 1):
        # Check if the substring matches the pattern
        if txt[i:i + m] == pat:
            print(f"Pattern found at index {i}")

# Example usage
txt = "AABAACAADAABAABA"
pat = "AABA"
search(pat, txt)
```

Output

Result:

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.

Aim

To implement and compare **Insertion Sort** and **Heap Sort** in Python, we will:

1. Implement both sorting algorithms.
2. Measure the time taken to sort different sizes of lists.
3. Plot a graph of the time taken versus the number of elements (n).

Algorithm for Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It is much like sorting playing cards in your hands.

Steps:

1. Start from the second element (since the first element is already considered sorted).
2. Compare the current element with the previous element.
3. If the current element is smaller, shift all the elements greater than the current element to the right.
4. Insert the current element into its correct position in the sorted portion of the list.
5. Repeat the process for each subsequent element until the entire list is sorted.

Algorithm for Heap Sort

Heap Sort is based on the **binary heap** data structure. It first builds a max-heap from the list and then repeatedly extracts the maximum element from the heap, placing it in the sorted part of the array.

Steps:

1. **Build a Max Heap**: Convert the array into a max heap. A max heap is a complete binary tree where the value of each node is greater than or equal to its children.
    - For each element, starting from the middle, apply the **heapify** process to ensure the heap property is maintained.
2. **Extract Maximum Element**: The root of the max heap contains the largest element. Swap it with the last element in the heap.
    - Reduce the heap size by 1 (ignoring the last element which is now in its correct position).
    - Apply the **heapify** process again to restore the heap property.
3. Repeat the extraction of the maximum element until the heap is empty. The elements are now sorted in ascending order.

Python Code

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

# Insertion Sort
def insertion_sort(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

# Heap Sort
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

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

    # Extract elements from heap one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap the root (max element) with the last element
        heapify(arr, i, 0)  # Heapify the reduced heap

# Function to measure time for sorting
def measure_time(sorting_function, n_values):
    times = []
    for n in n_values:
        # Generate a random list of size n
        arr = [random.randint(1, 10000) for _ in range(n)]

        # Measure time for sorting
        start_time = time.time()
        sorting_function(arr)
        end_time = time.time()

        times.append(end_time - start_time)  # Append time taken
    return times

# Define list sizes
n_values = [100, 500, 1000, 5000, 10000, 20000, 50000]

# Measure time for both sorting algorithms
insertion_sort_times = measure_time(insertion_sort, n_values)
heap_sort_times = measure_time(heap_sort, n_values)

# Plotting the graph
plt.plot(n_values, insertion_sort_times, marker='o', label="Insertion Sort", color='r')
plt.plot(n_values, heap_sort_times, marker='o', label="Heap Sort", color='b')
```

```
plt.title("Sorting Time vs List Size")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.legend()
plt.grid(True)
plt.show()
```

Output:

Result:

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

Aim:

To implement **Breadth-First Search (BFS)** for graph traversal. BFS explores all the vertices of a graph level by level, starting from the source node. This algorithm is used to find the shortest path in an unweighted graph and is particularly useful for tasks like shortest path search, web crawlers, and more.

Algorithm:

1. **Initialization**:
    - Start from a source node.
    - Use a **queue** to keep track of the nodes to be visited.
    - Use a **visited list/set** to keep track of visited nodes to avoid revisiting.
2. **Traversal**:
    - Enqueue the source node.
    - While the queue is not empty:
        - Dequeue a node from the front of the queue.
        - Visit all its unvisited neighbors, mark them as visited, and enqueue them.
3. **Termination**:
    - The BFS will terminate when all reachable nodes have been visited.

BFS Algorithm:

1. **Start** from a source node.
2. **Mark** the source node as visited.
3. **Add** the source node to the queue.
4. While the queue is not empty:
    - **Dequeue** a node from the queue.
    - Visit all its **unvisited neighbors**, mark them as visited, and **enqueue** them.
5. Repeat until all nodes are visited or the queue is empty.

Python Code Implementation for BFS

```
from collections import deque

# BFS function to traverse the graph
def bfs(graph, start):
    # Initialize a set to track visited nodes
    visited = set()

    # Create a queue to manage the BFS traversal
    queue = deque([start])

    # Mark the start node as visited
    visited.add(start)

    # Traverse the graph
    while queue:
        # Dequeue a node from the queue
        node = queue.popleft()
        print(node, end=" ")

        # Visit all unvisited neighbors of the node
        for neighbor in graph[node]:
            if neighbor not in visited:
```

```
            visited.add(neighbor)
            queue.append(neighbor)

# Example usage of the BFS function

# Graph representation using adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Starting BFS from node 'A'
print("BFS Traversal starting from node 'A':")
bfs(graph, 'A')
```

Output:

Result:

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

Aim:

To implement **Depth-First Search (DFS)** for graph traversal. DFS explores a graph by starting at the root (or any arbitrary node) and explores as far as possible along each branch before backtracking. DFS is often used in tasks such as pathfinding, topological sorting, and cycle detection in graphs.

Algorithm:

1. **Initialization**:
   - Start from a given node.
   - Use a **stack** (or recursion) to manage the traversal process.
   - Use a **visited list/set** to track which nodes have already been visited.
2. **Traversal**:
   - Start with the source node, mark it as visited, and push it onto the stack (or use recursion).
   - Pop the node from the stack and visit all its unvisited neighbors.
   - For each unvisited neighbor, mark it as visited and push it onto the stack (or call DFS recursively).
3. **Termination**:
   - The DFS traversal terminates when all nodes are visited, or if there are no unvisited neighbors left in the stack.

DFS Algorithm (Recursive Version):

1. **Start** from a given node.
2. **Mark** the current node as visited.
3. For each neighbor of the current node:
   - If the neighbor is unvisited, visit it.
4. Repeat the above steps until all reachable nodes are visited.

Python Code Implementation for DFS

```
# DFS function to traverse the graph recursively
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()  # Initialize visited set if not provided
    # Mark the current node as visited
    visited.add(node)
    print(node, end=" ")
    # Recurse for all unvisited neighbors of the current node
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
# Example usage of the DFS function
# Graph representation using adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
```

```
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Starting DFS from node 'A'
print("DFS Traversal starting from node 'A':")
dfs(graph, 'A')
```

Output:

Result:

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.

Aim:
To implement **Dijkstra's Algorithm** to find the shortest paths from a given vertex to all other vertices in a weighted connected graph. Dijkstra's algorithm is used to find the shortest path in a graph where all the edge weights are non-negative.

Algorithm:
Dijkstra's algorithm works by repeatedly selecting the vertex with the smallest tentative distance, updating the distances to its neighbors, and marking it as visited. The process continues until all vertices have been visited or the shortest path to every vertex has been determined.

Dijkstra's Algorithm Steps:

1. **Initialization**:
    - Set the distance to the source vertex as 0 and the distance to all other vertices as infinity.
    - Use a priority queue (or min-heap) to efficiently retrieve the vertex with the smallest tentative distance.
2. **Processing**:
    - Start with the source vertex. For the current vertex, examine all its unvisited neighbors.
    - For each neighbor, calculate the tentative distance through the current vertex and update it if it's smaller than the current known distance.
    - Once a vertex's shortest distance is determined, mark it as visited (processed).
3. **Termination**:
    - The algorithm terminates when all vertices have been processed, meaning the shortest paths to all vertices have been found.

Python Code Implementation for Dijkstra's Algorithm:

```
import heapq  # To implement priority queue (min-heap)

# Function to implement Dijkstra's algorithm
def dijkstra(graph, start):
    # Initialize distances with infinity, except for the start vertex
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0

    # Min-heap priority queue to store vertices and their current shortest distance
    priority_queue = [(0, start)]  # (distance, vertex)

    while priority_queue:
        # Get the vertex with the smallest tentative distance
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # Skip if the vertex has already been processed
        if current_distance > distances[current_vertex]:
            continue

        # Explore each neighbor of the current vertex
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            # If a shorter path is found, update the distance and add to the queue
```

```
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage of the Dijkstra's algorithm function

# Graph representation using adjacency list with weights
graph = {
    'A': {'B': 4, 'C': 2},
    'B': {'A': 4, 'C': 5, 'D': 10},
    'C': {'A': 2, 'B': 5, 'D': 3},
    'D': {'B': 10, 'C': 3}
}

# Start Dijkstra's algorithm from vertex 'A'
start_vertex = 'A'
shortest_paths = dijkstra(graph, start_vertex)

# Print the shortest distances from the start vertex
print(f"Shortest paths from vertex '{start_vertex}':")
for vertex, distance in shortest_paths.items():
    print(f"Distance from {start_vertex} to {vertex}: {distance}")
```

Output:

Result:

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

Aim:
To implement **Prim's Algorithm** to find the **Minimum Cost Spanning Tree (MST)** of a given undirected graph. A Minimum Cost Spanning Tree connects all vertices in the graph with the minimum possible total edge weight, and Prim's algorithm is one of the greedy algorithms used to find it.

Algorithm:
Prim's algorithm works by starting with an arbitrary node and gradually expanding the MST by adding the smallest edge connecting a vertex in the MST to a vertex outside the MST. This process continues until all vertices are included in the MST.

Prim's Algorithm Steps:

1. **Initialization**:
    - Start from an arbitrary vertex, mark it as part of the MST.
    - Initialize a priority queue (or min-heap) to select the edge with the smallest weight at each step.
    - Set the initial key value (cost) of the starting vertex as 0, and the others as infinity.
2. **Processing**:
    - At each step, pick the vertex with the smallest edge cost that connects a vertex in the MST to a vertex outside the MST.
    - Mark the selected vertex as visited (part of the MST).
    - Update the key values for all its unvisited neighbors.
3. **Termination**:
    - The algorithm terminates when all vertices have been included in the MST.

Python Code Implementation for Prim's Algorithm:

```
import heapq  # To implement priority queue (min-heap)

# Function to implement Prim's algorithm for Minimum Cost Spanning Tree
def prim_mst(graph):
    # Initialize structures
    mst = []  # List to store edges in the MST
    visited = set()  # To track the visited vertices
    min_heap = [(0, list(graph.keys())[0])]  # (cost, vertex), start from any vertex

    while min_heap:
        # Get the vertex with the smallest edge cost
        cost, vertex = heapq.heappop(min_heap)

        # If the vertex has already been visited, skip it
        if vertex in visited:
            continue

        # Mark the vertex as visited
        visited.add(vertex)

        # Add the edge to the MST (we don't store the starting vertex cost)
        if cost != 0:
            mst.append((prev_vertex, vertex, cost))
```

```
        # Explore the neighbors of the current vertex
        for neighbor, weight in graph[vertex].items():
            if neighbor not in visited:
                heapq.heappush(min_heap, (weight, neighbor))
                prev_vertex = vertex  # Store the current vertex as the previous vertex

    return mst

# Example usage of the Prim's algorithm function

# Graph representation using adjacency list with weights
graph = {
    'A': {'B': 4, 'C': 2},
    'B': {'A': 4, 'C': 5, 'D': 10},
    'C': {'A': 2, 'B': 5, 'D': 3},
    'D': {'B': 10, 'C': 3}
}

# Find the Minimum Spanning Tree (MST) starting from any vertex (e.g., 'A')
mst = prim_mst(graph)

# Print the edges of the MST with their costs
print("Edges in the Minimum Spanning Tree (MST):")
total_cost = 0
for edge in mst:
    print(f"{edge[0]} - {edge[1]} with cost {edge[2]}")
    total_cost += edge[2]

print(f"\nTotal cost of the Minimum Spanning Tree: {total_cost}")
```

Output:

Result:

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

Aim:

To implement **Floyd's Algorithm** for the **All-Pairs Shortest Paths** problem. The goal of the algorithm is to find the shortest paths between all pairs of vertices in a given weighted graph. Floyd's algorithm is an example of a dynamic programming approach, where intermediate vertices are considered to find the shortest paths between all pairs of vertices.

Algorithm:

Floyd's algorithm computes the shortest paths between all pairs of vertices by considering every possible intermediate vertex. The algorithm iteratively improves the path between two vertices by introducing a new intermediate vertex.

Floyd's Algorithm Steps:

1. **Initialization**:
    - Start with a distance matrix where each entry dist[i][j] represents the weight of the edge between vertices i and j. If there is no edge, initialize the distance as infinity.
    - Set the diagonal elements (distance from a vertex to itself) to 0.
2. **Iterate Over All Pairs of Vertices**:
    - For each intermediate vertex k, and for each pair of vertices i and j, check if the path from i to j through k is shorter than the current known path from i to j.
    - Update the distance matrix accordingly.
3. **Termination**:
    - The algorithm terminates when all possible intermediate vertices have been considered, and the distance matrix contains the shortest path distances between all pairs of vertices.

Python Code Implementation for Floyd's Algorithm:

```
# Function to implement Floyd's Algorithm for All-Pairs Shortest Paths
def floyd_warshall(graph):
    # Step 1: Initialize the distance matrix
    # Start with infinity for all distances except diagonal (which is 0)
    vertices = list(graph.keys())
    dist = {vertex: {v: float('inf') for v in vertices} for vertex in vertices}

    for vertex in vertices:
        dist[vertex][vertex] = 0  # Distance from a vertex to itself is 0

    # Step 2: Set the initial distances based on the given graph
    for vertex in graph:
        for neighbor, weight in graph[vertex].items():
            dist[vertex][neighbor] = weight

    # Step 3: Apply Floyd's Algorithm (Dynamic Programming)
    for k in vertices:
        for i in vertices:
            for j in vertices:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Example usage of the Floyd-Warshall function
```

```
# Graph representation using adjacency list with weights
graph = {
    'A': {'B': 4, 'C': 2},
    'B': {'A': 4, 'C': 5, 'D': 10},
    'C': {'A': 2, 'B': 5, 'D': 3},
    'D': {'B': 10, 'C': 3}
}

# Run Floyd-Warshall algorithm to find shortest paths between all pairs of vertices
shortest_paths = floyd_warshall(graph)

# Print the resulting shortest path matrix
print("Shortest paths between all pairs of vertices:")
for vertex in shortest_paths:
    print(f"{vertex}: {shortest_paths[vertex]}")
```

Output:

Result:

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

Aim:

To compute the **Transitive Closure** of a given **directed graph** using **Warshall's Algorithm**. The transitive closure of a graph is a matrix that indicates whether there is a path between any two vertices in the graph. Warshall's algorithm helps in finding this transitive closure by iteratively updating the reachability between pairs of vertices.

Algorithm:

Warshall's algorithm computes the transitive closure of a directed graph by using a dynamic programming approach. It systematically updates a reachability matrix to determine whether a path exists between any two vertices.

Warshall's Algorithm Steps:

1. **Initialization**:
    - Start with an **adjacency matrix** that represents the graph. The matrix reach[i][j] is set to 1 if there is a direct edge from vertex i to vertex j and 0 otherwise. For self-loops (i.e., i == j), the diagonal elements are set to 1.
2. **Iterative Process**:
    - For each vertex k, check if there is a path from vertex i to vertex j through vertex k. If reach[i][k] and reach[k][j] are both 1, then set reach[i][j] = 1. This step is repeated for all vertices k, i, and j.
3. **Termination**:
    - The algorithm terminates when all vertices have been processed, and the reachability matrix contains the transitive closure of the graph.

Python Code Implementation for Warshall's Algorithm:

```
# Function to compute the transitive closure using Warshall's Algorithm
def warshall(graph):
    # Step 1: Initialize the adjacency matrix for reachability
    vertices = list(graph.keys())
    n = len(vertices)

    # Initialize the reachability matrix (reach[i][j] = 1 if there is a direct edge, otherwise 0)
    reach = [[0] * n for _ in range(n)]

    # Create a mapping from vertices to indices
    vertex_to_index = {vertex: idx for idx, vertex in enumerate(vertices)}

    # Set reachability based on direct edges in the graph
    for i in range(n):
        reach[i][i] = 1  # A vertex is reachable from itself
        for neighbor in graph[vertices[i]]:
            reach[i][vertex_to_index[neighbor]] = 1  # Direct edge from i to neighbor

    # Step 2: Apply Warshall's algorithm (update reachability)
    for k in range(n):  # Check intermediate vertex k
        for i in range(n):  # Check start vertex i
            for j in range(n):  # Check end vertex j
                # If there is a path from i to k and from k to j, update reachability
                if reach[i][k] == 1 and reach[k][j] == 1:
                    reach[i][j] = 1
```

```
    return reach

# Function to print the transitive closure matrix
def print_transitive_closure(reach):
    print("Transitive Closure Matrix:")
    for row in reach:
        print(row)

# Example usage of Warshall's algorithm

# Graph representation using adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['C'],
    'C': ['D'],
    'D': []
}

# Compute the transitive closure
transitive_closure = warshall(graph)

# Print the transitive closure matrix
print_transitive_closure(transitive_closure)
```

Output:

Result:

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.

Aim:

To find the **maximum** and **minimum** numbers in a given list of **n** numbers using the **Divide and Conquer** technique. The goal is to break the problem down into smaller subproblems, find the maximum and minimum in those subproblems, and then combine the results to find the overall maximum and minimum.

Algorithm:

The divide and conquer technique involves breaking the list into smaller sublists, finding the minimum and maximum in each sublist, and then merging the results to get the overall minimum and maximum.

Steps for Divide and Conquer Approach:

1. **Base Case**:
    - If the list contains only one element, that element is both the minimum and maximum.
    - If the list contains two elements, compare them directly to determine the minimum and maximum.
2. **Divide**:
    - Divide the list into two halves.
3. **Conquer**:
    - Recursively find the minimum and maximum in both halves.
4. **Combine**:
    - Once the minimum and maximum of the two halves are found, compare them to get the overall minimum and maximum for the entire list.

Python Code Implementation:

```
# Function to find the minimum and maximum using Divide and Conquer
def find_min_max(arr, low, high):
    # Base Case: If the list contains only one element
    if low == high:
        return arr[low], arr[low]

    # Base Case: If the list contains two elements
    elif high == low + 1:
        if arr[low] < arr[high]:
            return arr[low], arr[high]
        else:
            return arr[high], arr[low]

    # Divide the list into two halves
    mid = (low + high) // 2

    # Recursively find the min and max in the left half and right half
    left_min, left_max = find_min_max(arr, low, mid)
    right_min, right_max = find_min_max(arr, mid + 1, high)

    # Combine the results and return the overall min and max
    overall_min = min(left_min, right_min)
    overall_max = max(left_max, right_max)

    return overall_min, overall_max
```

```
# Example usage of the function
arr = [12, 3, 5, 7, 19, 1, 10]
n = len(arr)

# Finding the minimum and maximum using divide and conquer technique
minimum, maximum = find_min_max(arr, 0, n - 1)

print("Minimum:", minimum)
print("Maximum:", maximum)
```

Output:

Result:

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.

Aim:
To implement the **Merge Sort** and **Quick Sort** algorithms to sort an array of elements and determine the time required to sort. The program will repeat the experiment for different values of **n** (the number of elements in the list) and plot a graph of the time taken versus **n**.

Algorithms:

Merge Sort Algorithm:
Merge Sort is a **divide and conquer** algorithm that divides the array into two halves, recursively sorts each half, and then merges the sorted halves.

Steps:

1. **Divide** the unsorted list into two halves.
2. **Recursively** divide each half until the subarrays contain a single element.
3. **Merge** the sorted subarrays to produce the sorted array.

Time Complexity: O(nlog⁡n)O(n \log n)O(nlogn) **Space Complexity**: O(n)O(n)O(n)

Quick Sort Algorithm:
Quick Sort is another **divide and conquer** algorithm. It works by selecting a "pivot" element, partitioning the array into two parts based on whether elements are smaller or larger than the pivot, and recursively sorting the subarrays.

Steps:

1. **Pick** a pivot element from the array.
2. **Partition** the array into two subarrays: elements smaller than the pivot and elements larger than the pivot.
3. **Recursively** apply the same process to the subarrays.

Time Complexity:

- **Best and Average Case**: O(nlog⁡n)O(n \log n)O(nlogn)
- **Worst Case**: O(n2)O(n^2)O(n2) (if the pivot selection is poor) **Space Complexity**: O(log⁡n)O(\log n)O(logn) (due to recursion stack)

Python Code Implementation:

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

# Merge Sort Function
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
    return arr

# Quick Sort Function
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Function to time sorting algorithms for different values of n
def time_sorting_algorithm(sort_function, arr):
    start_time = time.time()
    sort_function(arr)
    end_time = time.time()
    return end_time - start_time

# Function to run experiment and plot time vs n
def run_experiment():
    n_values = []
    merge_sort_times = []
    quick_sort_times = []

    for n in range(100, 1001, 100):  # Experiment with different n values (100, 200, ..., 1000)
        arr = [random.randint(1, 10000) for _ in range(n)]

        # Measure time for Merge Sort
        arr_copy = arr.copy()
        merge_time = time_sorting_algorithm(merge_sort, arr_copy)
        merge_sort_times.append(merge_time)

        # Measure time for Quick Sort
        arr_copy = arr.copy()
        quick_time = time_sorting_algorithm(quick_sort, arr_copy)
        quick_sort_times.append(quick_time)
```

```
        n_values.append(n)

    # Plotting the results
    plt.figure(figsize=(10, 5))
    plt.plot(n_values, merge_sort_times, label="Merge Sort", color='blue')
    plt.plot(n_values, quick_sort_times, label="Quick Sort", color='green')
    plt.xlabel('Number of elements (n)')
    plt.ylabel('Time taken (seconds)')
    plt.title('Sorting Algorithm Time Comparison')
    plt.legend()
    plt.grid(True)
    plt.show()

# Run the experiment and plot the graph
run_experiment()
```

Output:

Result:

13. Implement N Queens problem using Backtracking.

Aim:
To solve the **N Queens** problem using the **Backtracking** technique. The goal is to place **N queens** on an **N×N chessboard** such that no two queens attack each other. This means no two queens can share the same row, column, or diagonal.

Algorithm:

1. **Backtracking Approach**:
    - Start placing queens in the first row and try to place them in each column of that row.
    - After placing a queen in a valid position, recursively try to place queens in the next row.
    - If a valid placement is found, move to the next row.
    - If no valid placement is found for a queen in a row, backtrack by removing the queen and trying the next column in the previous row.
    - The algorithm continues until all queens are placed on the board or all possible positions are explored.
2. **Valid Position**:
    - A queen cannot share the same column as another queen.
    - A queen cannot share the same diagonal (both principal and secondary diagonals) with another queen.
3. **Base Case**: If we have successfully placed queens in all rows, the solution is found.
4. **Recursive Case**: Try placing queens row by row, checking for valid placements, and backtrack when necessary.

Steps for Backtracking:

1. Place a queen in the first available column of a row.
2. Move to the next row and attempt to place a queen.
3. If placing a queen in a column causes a conflict (same column or diagonal), try the next column.
4. If all rows are filled successfully, return the solution.
5. If no placement is possible for a row, backtrack and move the queen to the previous row.

Python Code Implementation:

```
# Function to check if a queen can be placed on board[row][col]
def is_safe(board, row, col, N):
    # Check this column on upper side
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row - 1, -1, -1), range(col - 1, -1, -1)):
        if board[i][j] == 1:
            return False
```

```
    # Check upper diagonal on right side
    for i, j in zip(range(row - 1, -1, -1), range(col + 1, N)):
        if board[i][j] == 1:
            return False

    return True

# Function to solve the N Queens problem using backtracking
def solve_n_queens(board, row, N):
    # If all queens are placed, return True
    if row >= N:
        return True

    # Try placing the queen in all columns for the current row
    for col in range(N):
        if is_safe(board, row, col, N):
            # Place the queen
            board[row][col] = 1

            # Recur to place the next queen
            if solve_n_queens(board, row + 1, N):
                return True

            # Backtrack if placing queen in current position does not work
            board[row][col] = 0

    return False

# Function to print the board
def print_board(board, N):
    for i in range(N):
        for j in range(N):
            if board[i][j] == 1:
                print("Q", end=" ")
            else:
                print(".", end=" ")
        print()

# Main function to solve the N Queens problem
def n_queens(N):
    board = [[0 for _ in range(N)] for _ in range(N)]

    if not solve_n_queens(board, 0, N):
        print("Solution does not exist")
        return False

    print_board(board, N)
    return True
```

```
# Example usage for N=4
N = 4
n_queens(N)
```

Output:

Result:

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.

Aim:
To solve the **Traveling Salesperson Problem (TSP)** by implementing both an **exact solution** (using a brute-force approach or dynamic programming) and an **approximation algorithm** (using the **Nearest Neighbor Algorithm**) and determine the error in the approximation.

TSP Problem Overview:
The Traveling Salesperson Problem (TSP) is a classic optimization problem where the objective is to find the shortest possible route that visits each city exactly once and returns to the starting point. Given a set of cities and the distances between every pair of cities, the goal is to determine the shortest path that visits all the cities exactly once.

Algorithms:

1. Exact Solution - Brute Force (or Dynamic Programming)

- **Brute Force Approach**:
    - Generate all possible permutations of cities and calculate the distance for each route. This ensures we find the exact optimal solution but is inefficient for large numbers of cities.
    - **Time Complexity**: O(n!)O(n!)O(n!), where nnn is the number of cities.
- **Dynamic Programming Approach** (for small nnn):
    - Use a **memoized recursive** approach to solve TSP efficiently in O(n2·2n)O(n^2 \cdot 2^n)O(n2·2n) time.

2. Approximation Solution - Nearest Neighbor Algorithm

- Start at any city, and at each step, move to the nearest unvisited city until all cities are visited.
- This algorithm provides a fast, greedy approximation to the problem but does not guarantee an optimal solution.
- **Time Complexity**: O(n2)O(n^2)O(n2), where nnn is the number of cities.

3. Error Determination

- After solving the TSP problem using both exact and approximation algorithms, we calculate the error:

$$\text{Error} = \frac{\text{Approximate Distance} - \text{Optimal Distance}}{\text{Optimal Distance}} \times 100$$

The error shows how close the approximation is to the optimal solution.

Python Code Implementation:

```
import itertools
import math
import random

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

# Brute force approach to find the optimal solution for TSP
def tsp_brute_force(cities):
    n = len(cities)
```

```
    min_path = None
    min_dist = float('inf')

    # Generate all possible permutations of the cities (except the starting city)
    for perm in itertools.permutations(range(1, n)):
        # Calculate the total distance of the path
        dist = 0
        prev_city = 0  # Starting from the first city
        for city in perm:
            dist += distance(cities[prev_city], cities[city])
            prev_city = city
        dist += distance(cities[prev_city], cities[0])  # Return to the starting city

        # Update the minimum distance and path if needed
        if dist < min_dist:
            min_dist = dist
            min_path = (0,) + perm + (0,)  # Adding start and end cities

    return min_dist, min_path

# Nearest Neighbor Approximation Algorithm
def nearest_neighbor(cities):
    n = len(cities)
    visited = [False] * n
    path = [0]  # Start from the first city
    visited[0] = True
    total_distance = 0

    for _ in range(n - 1):
        last_city = path[-1]
        nearest_city = None
        min_dist = float('inf')

        # Find the nearest unvisited city
        for city in range(n):
            if not visited[city]:
                dist = distance(cities[last_city], cities[city])
                if dist < min_dist:
                    min_dist = dist
                    nearest_city = city

        path.append(nearest_city)
        visited[nearest_city] = True
        total_distance += min_dist

    # Add the distance to return to the starting city
    total_distance += distance(cities[path[-1]], cities[0])

    return total_distance, path

# Function to calculate error in approximation
def calculate_error(optimal_distance, approx_distance):
    return ((approx_distance - optimal_distance) / optimal_distance) * 100

# Main function to test the TSP with both exact and approximation algorithms
```

```
def solve_tsp():
    # Random cities (x, y) coordinates for the problem instance
    cities = [(random.randint(0, 10), random.randint(0, 10)) for _ in range(5)]

    # Solve TSP using Brute Force (Exact solution)
    optimal_distance, optimal_path = tsp_brute_force(cities)
    print("Optimal Path (Brute Force):", optimal_path)
    print("Optimal Distance (Brute Force):", optimal_distance)

    # Solve TSP using Nearest Neighbor (Approximation)
    approx_distance, approx_path = nearest_neighbor(cities)
    print("Approximate Path (Nearest Neighbor):", approx_path)
    print("Approximate Distance (Nearest Neighbor):", approx_distance)

    # Calculate the error between the approximation and the optimal solution
    error = calculate_error(optimal_distance, approx_distance)
    print("Error in Approximation: {:.2f}%".format(error))

# Run the TSP solver
solve_tsp()
```

Output:

Result:

15. Implement randomized algorithms for finding the kth smallest number. The programs can be implemented in C/C++/JAVA/ Python.

Aim:

To implement **randomized algorithms** for finding the **k-th smallest element** in an unordered list of numbers. The primary goal is to use a random pivot-based strategy (like **Quickselect**) to efficiently find the k-th smallest element in an average time complexity of **O(n)**.

Algorithm:

1. **Quickselect Algorithm**:
    - **Quickselect** is an efficient algorithm to find the k-th smallest element in an unsorted list. It is based on the **Quicksort** algorithm and uses the **partitioning** technique to narrow down the search space.
    - The idea is to select a pivot randomly, partition the array around the pivot, and decide whether the k-th smallest element lies on the left or right of the pivot. If it is on the left, recursively search the left half, otherwise, search the right half.
    - This algorithm works in **O(n)** on average, but in the worst case, it takes **$O(n^2)$**.

Steps:

1. Randomly pick a pivot from the array.
2. Partition the array into two halves:
    - Elements less than the pivot.
    - Elements greater than the pivot.
3. Check the position of the pivot:
    - If the pivot's index matches the k-th position, return the pivot.
    - If the k-th element is smaller, recursively search the left partition.
    - If the k-th element is larger, recursively search the right partition.

Python Code Implementation:

```
import random

# Function to partition the array around the pivot
def partition(arr, low, high):
    # Choose a random pivot index
    pivot_index = random.randint(low, high)
    pivot_value = arr[pivot_index]

    # Swap the pivot with the last element
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

    # Partition the array
    i = low
    for j in range(low, high):
        if arr[j] < pivot_value:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1

    # Place the pivot in its correct position
    arr[i], arr[high] = arr[high], arr[i]
```

```
    return i  # Return the index of the pivot

# Function to find the k-th smallest element using Quickselect
def quickselect(arr, low, high, k):
    if low == high:  # Base case: only one element
        return arr[low]

    # Partition the array and get the pivot index
    pivot_index = partition(arr, low, high)

    # The position of the pivot in the sorted array is pivot_index
    if k == pivot_index:
        return arr[k]
    elif k < pivot_index:
        return quickselect(arr, low, pivot_index - 1, k)
    else:
        return quickselect(arr, pivot_index + 1, high, k)

# Main function to find the k-th smallest element
def find_kth_smallest(arr, k):
    if k < 1 or k > len(arr):
        raise ValueError("k must be between 1 and the length of the array.")

    # Convert k to 0-based index
    return quickselect(arr, 0, len(arr) - 1, k - 1)

# Example usage
arr = [12, 3, 5, 7, 19, 2, 11, 17]
k = 4  # Find the 4th smallest element
result = find_kth_smallest(arr, k)
print(f"The {k}-th smallest element is: {result}")
```

Output:

Result: