



Affiliated To Anna University, Chennai & Approved by AICTE, New Delhi.

Coimbatore, Tamil Nadu, India – 641 021



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (REGIONAL)

22CS402R ALGORITHMS

LAB RECORD

NAME :

BRANCH :

REGISTER NUMBER :

YEAR / SEMESTER :

ACADEMIC YEAR :

SUBJECT CODE :

SUBJECT NAME :



Affiliated To Anna University, Chennai & Approved by AICTE, New Delhi.

Coimbatore, Tamil Nadu, India – 641 021

BONAFIDE CERTIFICATE

NAME :

ACADEMIC YEAR :

YEAR/SEMESTER :

BRANCH :

UNIVERSITY REGISTER NUMBER:

Certified that this is the bonafide record of work done by the above student in the

_____ Laboratory during the year 2024-2025.

Staff-in-Charge

Head of the Department

Submitted for the Practical Examination held on

Internal Examiner

External Examiner

INDEX

[illegible]

[illegible]

EX.NO:1
DATE:

IMPLEMENT LINEAR SEARCH

AIM:

The aim of linear search is to find a specific element in a list by checking each element one by one until the target element is found or the end of the list is reached and 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. Start from the beginning of the list.
2. Compare the target element with each element in the list.
3. If the target element is found, return the index.
4. If the end of the list is reached without finding the target element, return -1.
5. Repeat the experiments for different values of n and draw graph

PROGRAM:

```
import time
import matplotlib.pyplot as plt
# Function to perform Linear Search
def linear_search(arr, target):
    """
    Perform a linear search to find the target in the given list.
    Returns the index if found, else -1.
    """
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1
# Function to measure the time taken for Linear Search
def measure_search_time(arr, target):
    start_time = time.time() # Record the start time
    linear_search(arr, target) # Perform the search
    end_time = time.time() # Record the end time
    return end_time - start_time # Calculate elapsed time
# Main program
if __name__ == "__main__":
    # List of sizes to test
    sizes = [1000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000]
    times = [] # To store time taken for each size
    # Loop over each size
    for n in sizes:
        # Generate a list of size n
        elements = list(range(1, n + 1)) # Sequential numbers
        target_value = n # Target to search for (worst case: last element)
        # Measure the time taken for the search
```

```
duration = measure_search_time(elements, target_value)
times.append(duration)
# Display results for each size
print(f"Size: {n}, Time Taken: {duration:.8f} seconds")
# Plot the graph of time taken vs list size
plt.figure(figsize=(10, 6))
plt.plot(sizes, times, marker='o', label='Linear Search Time')
plt.title('Linear Search Time vs List Size (n)')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time Taken (seconds)')
plt.grid()
plt.legend()
plt.show()
```

OUTPUT:

Size: 1000, Time Taken: 0.00088429 seconds

Size: 5000, Time Taken: 0.00037622 seconds

Size: 10000, Time Taken: 0.00074172 seconds

Size: 20000, Time Taken: 0.00289321 seconds

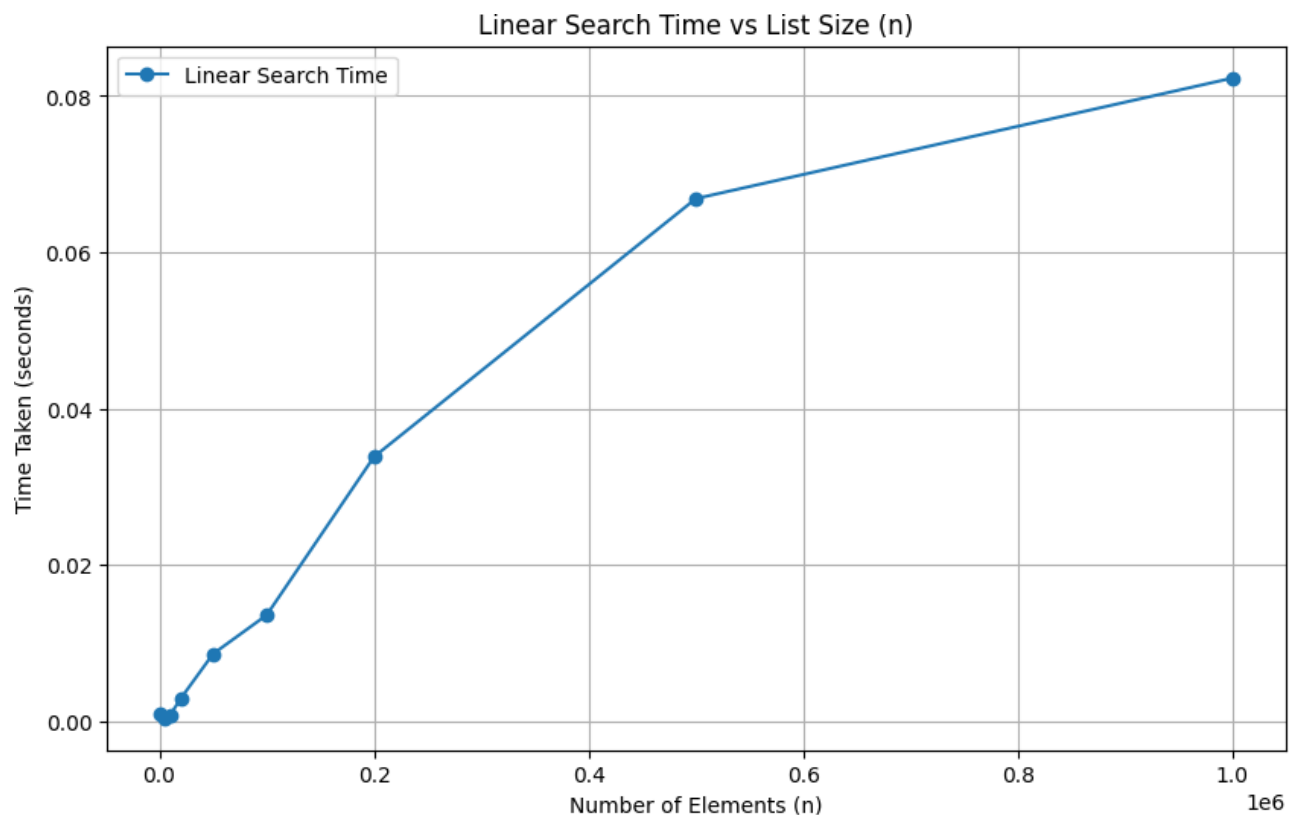
Size: 50000, Time Taken: 0.00858378 seconds

Size: 100000, Time Taken: 0.01355886 seconds

Size: 200000, Time Taken: 0.03380585 seconds

Size: 500000, Time Taken: 0.06685400 seconds

Size: 1000000, Time Taken: 0.08227754 seconds



RESULT:

Thus, aim of linear search is to find a specific element in a list by checking each element one by one until the target element is found or the end of the list is reached and plot the graph with different values of n.

EX.NO:2

IMPLEMENT RECURSIVE BINARY SEARCH

DATE:

AIM:

To search for a target element in a sorted array using recursive Binary Search and 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:

1. Compare the target element with the middle element of the array.
2. If the target element is equal to the middle element, return the index.
3. If the target element is less than the middle element, recursively search the left half of the array.
4. If the target element is greater than the middle element, recursively search the right half of the array.
5. If the array is empty or the target element is not present, return -1.
6. Repeat the experiments for different values of n and draw graph

PROGRAM:

```
import time
import matplotlib.pyplot as plt

# Recursive Binary Search Implementation
def binary_search_recursive(arr, target, low, high):
    if low > high:
        return -1 # Element not found
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, high)
    else:
        return binary_search_recursive(arr, target, low, mid - 1)

# Measure time for binary search on different list sizes
def measure_search_time():
    times = []
    n_values = [10**i for i in range(1, 7)] # List sizes: 10, 100, 1000, ..., 1000000
    for n in n_values:
        arr = list(range(n)) # Sorted list
        target = n - 1 # Element to search (last element)
        start_time = time.time()
        binary_search_recursive(arr, target, 0, n - 1)
        end_time = time.time()
        times.append(end_time - start_time)
    return n_values, times

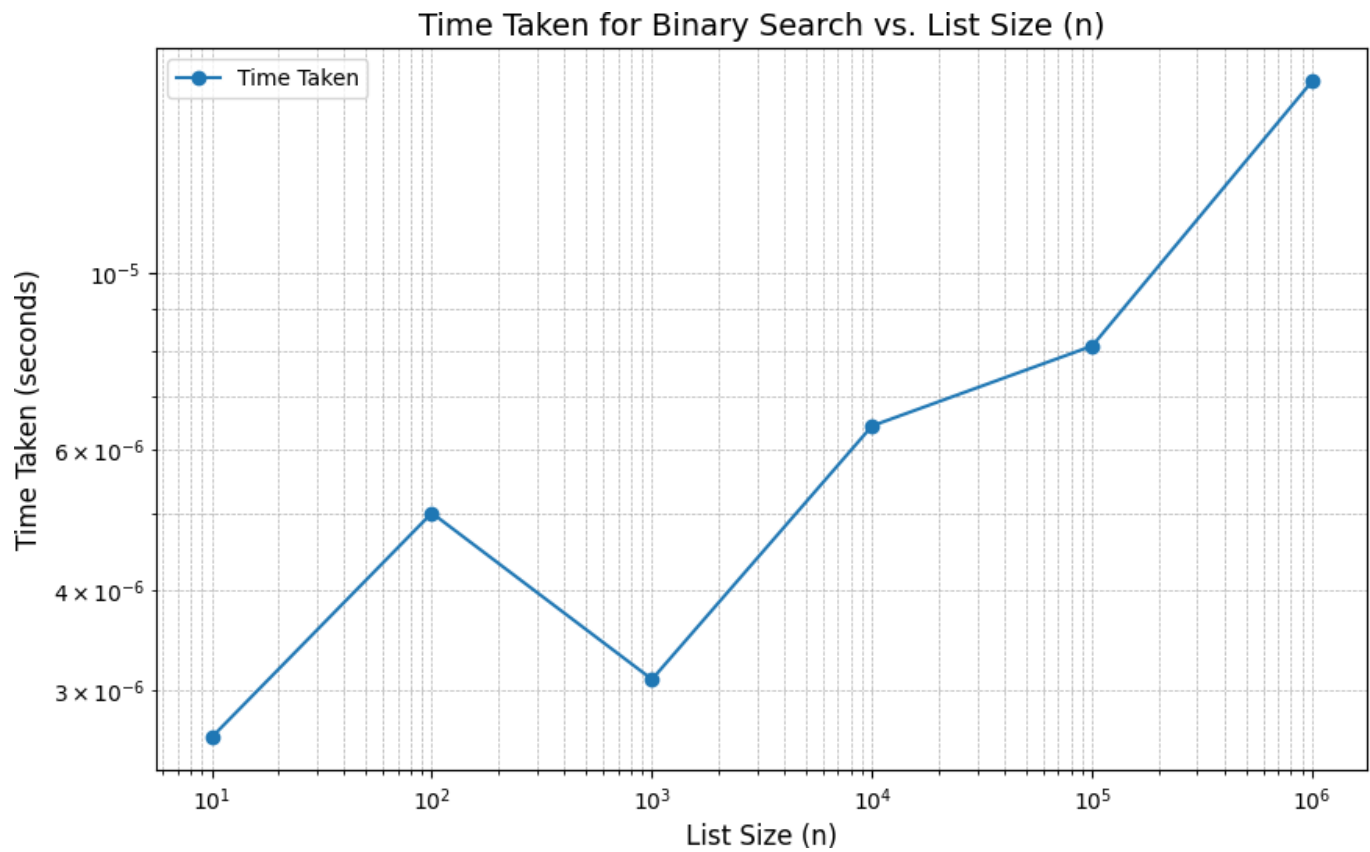
# Plotting the results
def plot_graph(n_values, times):
    plt.figure(figsize=(10, 6))
```



```
plt.plot(n_values, times, marker='o', label='Time Taken')
plt.title('Time Taken for Binary Search vs. List Size (n)', fontsize=14)
plt.xlabel('List Size (n)', fontsize=12)
plt.ylabel('Time Taken (seconds)', fontsize=12)
plt.xscale('log')
plt.yscale('log')
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.legend()
plt.show()
```

```
# Main Execution
n_values, times = measure_search_time()
plot_graph(n_values, times)
```

OUTPUT:



RESULT:

Thus, to search for a target element in a sorted array using recursive Binary Search is executed successfully and its output verified and plot the graph with different values of n .

EX.NO:3	NAIVE PATTERN SEARCHING ALGORITHM
DATE:	

AIM:

Implement the Naive Pattern Searching algorithm to find all occurrences of a pattern in a text.

PROCEDURE:

The Naive Pattern Searching algorithm involves iterating through the text and checking if the pattern matches at each position. If a match is found, we record the position or print it.

1. Start from the first character of the text and iterate through each character.
2. For each position, compare the characters of the pattern with the characters in the text, starting from that position.
3. If a match is found, print or record the position of the match.
4. Continue the process until the end of the text is reached.

PROGRAM:

```
def naive_pattern_search(text, pattern):
    text_length = len(text)
    pattern_length = len(pattern)
    occurrences = []
    for i in range(text_length - pattern_length + 1):
        match = True
        for j in range(pattern_length):
            if text[i + j] != pattern[j]:
                match = False
                break
        if match:
            occurrences.append(i)
    return occurrences

# Example
text_input = "ABABCABABABCABABAB"
pattern_input = "ABAB"
result = naive_pattern_search(text_input, pattern_input)
print(f"Text: {text_input}")
print(f"Pattern: {pattern_input}")
print(f"Occurrences found at positions: {result}")
```

OUTPUT:



Text: ABABCABABABCABABAB

Pattern: ABAB

Occurrences found at positions: [0, 5, 7, 12, 14]

RESULT:

Thus implement the Naive Pattern Searching algorithm to find all occurrences of a pattern in a text is executed successfully and its output verified.

EX.NO:4	INSERTION SORT AND HEAP SORT
DATE:	

AIM:

To implement the Insertion sort and Heap sort algorithm using Python. The aim of Heap Sort is to sort an array in ascending or descending order using a binary heap data structure. 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:

1. Start with the second element (index 1) and consider it as the key.
2. Compare the key with the element before it.
3. If the key is smaller, move the larger element to the right.
4. Repeat step 3 until a smaller element is found or the beginning of the array is reached.
5. Insert the key at the correct position.
6. Repeat steps 1-5 for the remaining elements in the array.

HEAP SORT:

1. Build a max heap from the given array.
2. Swap the root (maximum element) with the last element of the heap and reduce the size of the heap by 1.
3. Heapify the root of the heap.
4. Repeat steps 2 and 3 until the heap is empty.

PROGRAM:

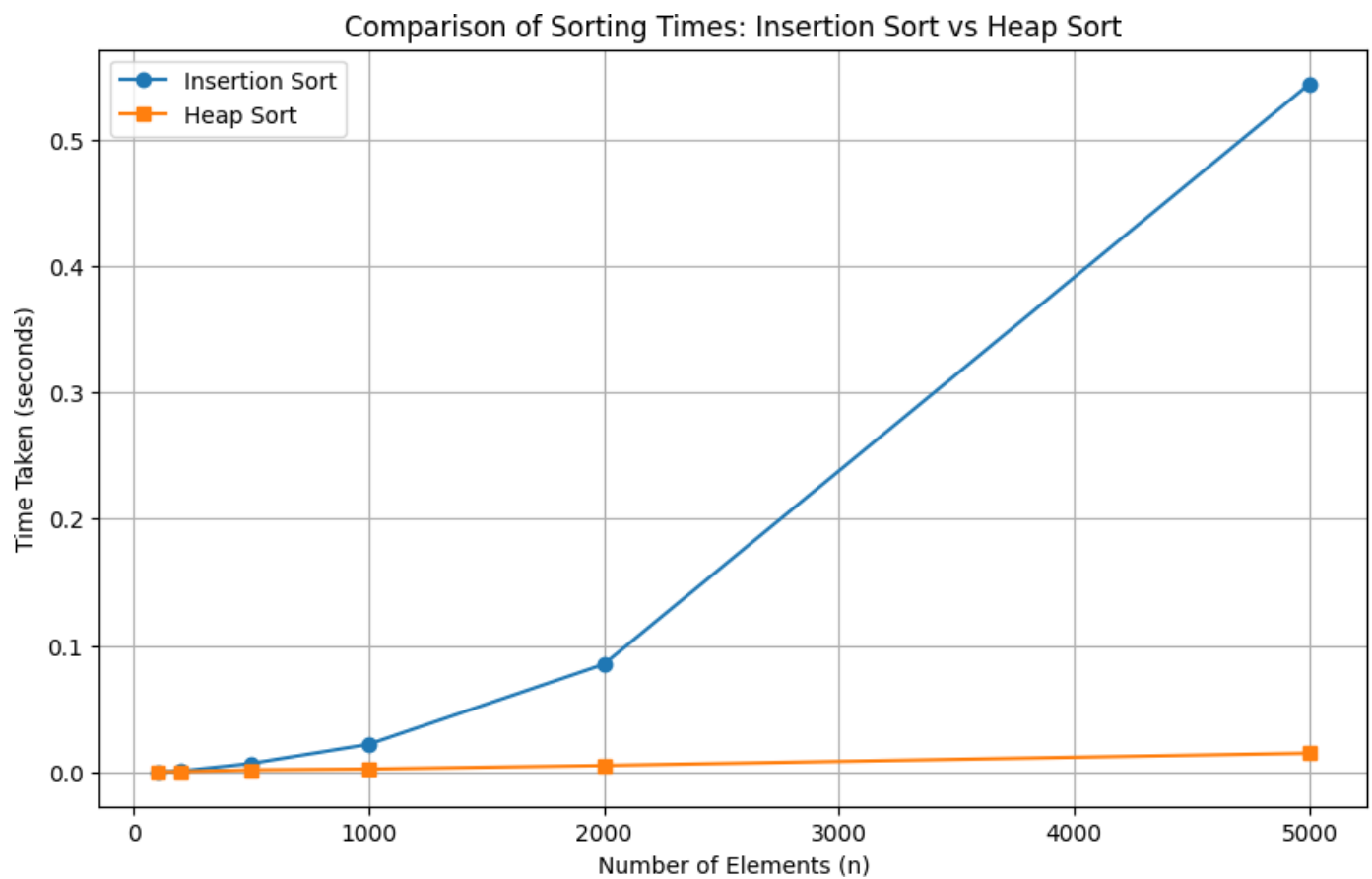
```
import time
import random
import matplotlib.pyplot as plt
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    def heapify(arr, n, i):
        largest = i # Initialize largest as root
        left = 2 * i + 1 # Left child
        right = 2 * i + 2 # Right child
        # Check if left child exists and is greater than root
        if left < n and arr[left] > arr[largest]:
            largest = left
        # Check if right child exists and is greater than largest so far
        if right < n and arr[right] > arr[largest]:
```

```

    largest = right
    # Swap and continue heapifying if root is not largest
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
def heap_sort(arr):
    n = len(arr)
    # Build 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)
# Experiment settings
sizes = [100, 200, 500, 1000, 2000, 5000] # Different values of n
insertion_times = []
heap_times = []
for size in sizes:
    # Generate a random list of the current size
    arr = [random.randint(1, 10000) for _ in range(size)]
    # Measure time for Insertion Sort
    arr_copy = arr.copy()
    start_time = time.time()
    insertion_sort(arr_copy)
    end_time = time.time()
    insertion_times.append(end_time - start_time)
    # Measure time for Heap Sort
    arr_copy = arr.copy()
    start_time = time.time()
    heap_sort(arr_copy)
    end_time = time.time()
    heap_times.append(end_time - start_time)
# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(sizes, insertion_times, label="Insertion Sort", marker="o")
plt.plot(sizes, heap_times, label="Heap Sort", marker="s")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.title("Comparison of Sorting Times: Insertion Sort vs Heap Sort")
plt.legend()
plt.grid()
plt.show()

```

OUTPUT:



RESULT:

Thus, implement the insertion sort and binary sort is executed successfully and its output verified and plot the graph with different values of n.

EX.NO:5	IMPLEMENT GRAPH TRAVERSAL USING BREADTH FIRST SEARCH
DATE:	

AIM:

The aim is to implement a graph traversal using Breadth-First Search (BFS) in Python.

ALGORITHM:

1. Start with a queue and enqueue the starting node.
2. Mark the starting node as visited.
3. While the queue is not empty: a. Dequeue a node from the queue. b. Process the dequeued node. c. Enqueue all adjacent nodes of the dequeued node that are not yet visited. d. Mark each visited adjacent node.
4. Repeat steps 3 until the queue is empty.

PROGRAM:

```

from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u) # Uncomment this line for an undirected graph
    def bfs(self, start):
        visited = set()
        queue = [start]
        visited.add(start)
        while queue:
            current_node = queue.pop(0)
            print(current_node, end=" ")
            for neighbor in self.graph[current_node]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)

# Example usage
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
g.add_edge(2, 5)
g.add_edge(3, 6)
print("BFS Traversal:")
g.bfs(1)

```


OUTPUT:



BFS Traversal:

1 2 3 4 5 6

RESULT:

Thus, implement a graph traversal using Breadth-First Search (BFS) in Python is executed successfully and its output verified.

EX.NO:6
DATE:

IMPLEMENT GRAPH TRAVERSAL USING DEPTH FIRST SEARCH

AIM:

Implement Depth First Search (DFS) for graph traversal._

ALGORITHM:

1. Create a stack to keep track of vertices.
2. Push the starting vertex onto the stack.
3. While the stack is not empty:
 - Pop a vertex from the stack.
 - If the vertex is not visited:
 - Mark it as visited.
 - Process the vertex (print or store it).
 - Push all unvisited neighbors of the vertex onto the stack.

PROGRAM:

```
from collections import defaultdict
```

```
class Graph:
```

```
    def __init__(self):
```

```
        self.graph = defaultdict(list)
```

```
    def add_edge(self, u, v):
```

```
        self.graph[u].append(v)
```

```
    def dfs(self, start):
```

```
        stack = [start]
```

```
        visited = set()
```

```
        while stack:
```

```
            vertex = stack.pop()
```

```
            if vertex not in visited:
```

```
                print(vertex, end=" ") # Process the vertex (you can also store it in a list)
```

```
                visited.add(vertex)
```

```
                # Push unvisited neighbors onto the stack
```

```
                stack.extend(neighbor for neighbor in self.graph[vertex] if neighbor not in visited)
```

```
# Example usage
```

```
if __name__ == "__main__":
```

```
    g = Graph()
```

```
    g.add_edge(0, 1)
```

```
    g.add_edge(0, 2)
```

```
    g.add_edge(1, 2)
```

```
    g.add_edge(2, 0)
```

```
    g.add_edge(2, 3)
```

```
    g.add_edge(3, 3)
```

```
    print("Depth First Traversal (starting from vertex 2):")
```

```
    g.dfs(2)
```

OUTPUT:

```
⇒ Depth First Traversal (starting from vertex 2):  
2 3 0 1
```

RESULT:

Thus, Implement Depth First Search (DFS) for graph traversal is executed Successfully and its output get verified.

EX.NO:7

DATE:

DEVELOP A PROGRAM TO FIND THE SHORTEST PATHS TO OTHER VERTICES USING DIJKSTRA'S ALGORITHM.

AIM:

Implement Dijkstra's algorithm to find the shortest path in a weighted graph.

ALGORITHM:

1. Dijkstra's algorithm is a greedy algorithm that finds the shortest path between two nodes in a weighted graph.
2. The algorithm maintains a set of vertices whose shortest distance from the source is known.
3. At each step, it selects the vertex with the smallest known distance, explores its neighbors, and updates their distances if a shorter path is found.

PROGRAM:

```
import heapq

class Graph:
    def __init__(self):
        self.vertices = set()
        self.edges = {}

    def add_vertex(self, value):
        self.vertices.add(value)
        self.edges[value] = []

    def add_edge(self, from_vertex, to_vertex, weight):
        self.edges[from_vertex].append((to_vertex, weight))
        self.edges[to_vertex].append((from_vertex, weight)) # For undirected graph

    def dijkstra(self, start):
        distances = {vertex: float('infinity') for vertex in self.vertices}
        distances[start] = 0
        priority_queue = [(0, start)]

        while priority_queue:
            current_distance, current_vertex = heapq.heappop(priority_queue)

            if current_distance > distances[current_vertex]:
                continue

            for neighbor, weight in self.edges[current_vertex]:
                distance = current_distance + weight
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))

        return distances
```

```
# Example usage:
graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_vertex("D")
graph.add_vertex("E")

graph.add_edge("A", "B", 2)
graph.add_edge("A", "C", 4)
graph.add_edge("B", "C", 1)
graph.add_edge("B", "D", 7)
graph.add_edge("C", "E", 3)
graph.add_edge("D", "E", 1)

start_vertex = "A"
result = graph.dijkstra(start_vertex)

# Output:
print(f"Shortest distances from {start_vertex}: {result}")
```

OUTPUT:



```
➞ Shortest distances from A: {'D': 7, 'C': 3, 'B': 2, 'E': 6, 'A': 0}
```

RESULT:

Thus, Implement Dijkstra's algorithm to find the shortest path in a weighted graph is executed successfully and its output get verified.

EX.NO:8

PRIM'S ALGORITHM

DATE:

AIM:

To find the minimum spanning tree (MST) of a connected, undirected graph using Prim's algorithm.

ALGORITHM:

1. Start with an arbitrary node as the initial vertex.
2. Create a priority queue to store edges with their weights.
3. Mark the chosen initial vertex as visited.
4. Add all edges connected to the initial vertex to the priority queue.
5. While the priority queue is not empty:
 - Dequeue the edge with the minimum weight.
 - If the destination vertex of the edge is not visited:
 - Mark the destination vertex as visited.
 - Add the edge to the minimum spanning tree.
 - Add all edges connected to the destination vertex that are not visited to the priority queue.
6. Repeat step 5 until all vertices are visited.

PROGRAM:

```
import heapq
```

```
class Graph:
```

```
    def __init__(self, vertices):  
        self.vertices = vertices  
        self.graph = [[] for _ in range(vertices)]
```

```
    def add_edge(self, u, v, weight):  
        self.graph[u].append((v, weight))  
        self.graph[v].append((u, weight))
```

```
    def prim_mst(self):  
        min_spanning_tree = []  
        visited = [False] * self.vertices  
        priority_queue = []
```

```
        # Start with the first vertex  
        heapq.heappush(priority_queue, (0, 0)) # (weight, vertex)
```

```
        while priority_queue:  
            weight, current_vertex = heapq.heappop(priority_queue)
```

```
            if not visited[current_vertex]:  
                visited[current_vertex] = True  
                min_spanning_tree.append((current_vertex, weight))
```

```
            for neighbor, edge_weight in self.graph[current_vertex]:
```

```

        if not visited[neighbor]:
            heapq.heappush(priority_queue, (edge_weight, neighbor))

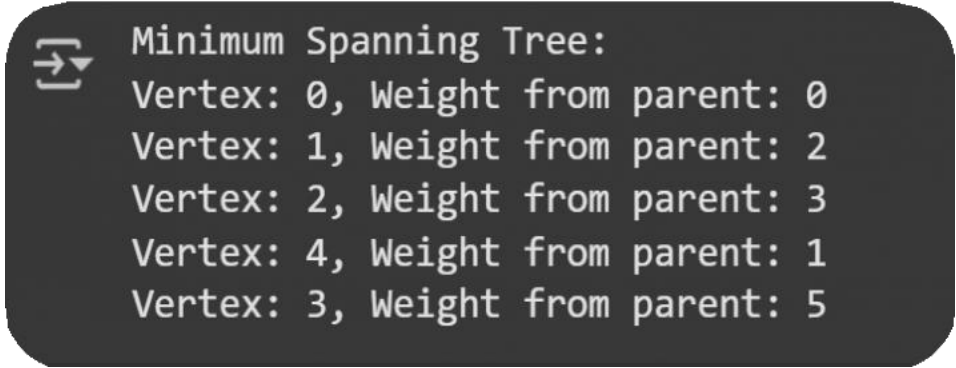
    return min_spanning_tree

# Example usage
g = Graph(5)
g.add_edge(0, 1, 2)
g.add_edge(0, 3, 8)
g.add_edge(1, 2, 3)
g.add_edge(1, 3, 6)
g.add_edge(2, 4, 1)
g.add_edge(3, 4, 5)

result = g.prim_mst()
print("Minimum Spanning Tree:")
for vertex, weight in result:
    print(f"Vertex: {vertex}, Weight from parent: {weight}")

```

OUTPUT:



```

➡ Minimum Spanning Tree:
Vertex: 0, Weight from parent: 0
Vertex: 1, Weight from parent: 2
Vertex: 2, Weight from parent: 3
Vertex: 4, Weight from parent: 1
Vertex: 3, Weight from parent: 5

```

RESULT:

Thus, to find the minimum spanning tree (MST) of a connected, undirected graph using Prim's algorithm is executed successfully and its output get verified.

EX.NO:9	FLOYD'S ALGORITHM
DATE:	

AIM:

Find the shortest paths between all pairs of vertices in a weighted graph

PROCEDURE:


The Floyd-Warshall algorithm works by considering all possible intermediate vertices in the paths between two vertices and gradually updates the shortest paths.

1. Initialize the distance matrix with the direct edges and set the distances to infinity where there is no direct edge.
2. For each intermediate vertex, update the distance matrix if a shorter path is found through that vertex.

PROGRAM:

```
INF = float('inf')
def floyd_warshall(graph):
    num_vertices = len(graph)
    distance_matrix = [[0 if i == j else graph[i][j] if graph[i][j] != 0 else INF for j in range(num_vertices)] for i
in range(num_vertices)]
    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if distance_matrix[i][k] + distance_matrix[k][j] < distance_matrix[i][j]:
                    distance_matrix[i][j] = distance_matrix[i][k] + distance_matrix[k][j]
    return distance_matrix
# Example usage:
graph = [
    [0, 3, INF, 5],
    [2, 0, INF, 4],
    [INF, 1, 0, INF],
    [INF, INF, 2, 0]
]
result = floyd_warshall(graph)
# Output the shortest distances between all pairs of vertices
for row in result:
    print(row)
```


OUTPUT:



```
[0, 3, 7, 5]
[2, 0, 6, 4]
[3, 1, 0, 5]
[5, 3, 2, 0]
```

RESULT:

Thus, find the shortest paths between all pairs of vertices in a weighted graph is executed successfully and its output get verified.

Ex.No:10	FIND OUT THE MAXIMUM AND MINIMUM NUMBERS USING DIVIDE AND CONQUER TECHNIQUE.
DATE:	

AIM:

To develop a Python program using the divide and conquer technique to find the maximum and minimum numbers in a given list of n number

ALGORITHM:

- **Base Case:**
 - If the list has only one element, return that element as both the maximum and minimum.
- **Divide:**
 - Split the list into two halves.
- **Conquer:**
 - Recursively find the maximum and minimum in each half.
- **Combine:**
 - Compare the maximum and minimum of the two halves to find the overall maximum and minimum.

PROGRAM:

```
def find_max_min(arr, start, end):
    # Base case: if there is only one element
    if start == end:
        return arr[start], arr[start]
    # If there are two elements, compare them and return
    if end - start == 1:
        return max(arr[start], arr[end]), min(arr[start], arr[end])
    # Divide the array into two halves
    mid = (start + end) // 2
    # Recursively find maximum and minimum in each half
    max1, min1 = find_max_min(arr, start, mid)
    max2, min2 = find_max_min(arr, mid + 1, end)
    # Combine the results
    overall_max = max(max1, max2)
    overall_min = min(min1, min2)
    return overall_max, overall_min

# Input list
numbers = [5, 3, 8, 2, 1, 7, 4, 6]
# Call the function
max_number, min_number = find_max_min(numbers, 0, len(numbers) - 1)
# Output
print(f"Given List: {numbers}")
print(f"Maximum Number: {max_number}")
print(f"Minimum Number: {min_number}")
```

OUTPUT:

```
➞ Given List: [5, 3, 8, 2, 1, 7, 4, 6]  
Maximum Number: 8  
Minimum Number: 1
```

RESULT:

Thus, to develop a Python program using the divide and conquer technique to find the maximum and minimum numbers in a given list of n number is executed successfully and its output get verified.

EX.NO:11	MERGE SORT AND QUICK SORT
DATE:	

AIM:

The aim of the Merge Sort algorithm is to efficiently sort an array or a list in ascending or descending order and Quick Sort algorithm is to efficiently sort an array or list of elements in ascending or descending order.

ALGORITHM:

MERGE SORT:

1. Divide the unsorted list into n sublists, each containing one element (base case).
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining.

QUICK SORT:

1. Choose a pivot element from the array. This can be done in various ways, such as picking the first, last, middle, or a random element.
2. Partition the array into two sub-arrays - elements less than the pivot and elements greater than the pivot.
3. Recursively apply the QuickSort algorithm to the sub-arrays.
4. Combine the sorted sub-arrays to get the final sorted array.

PROGRAM:

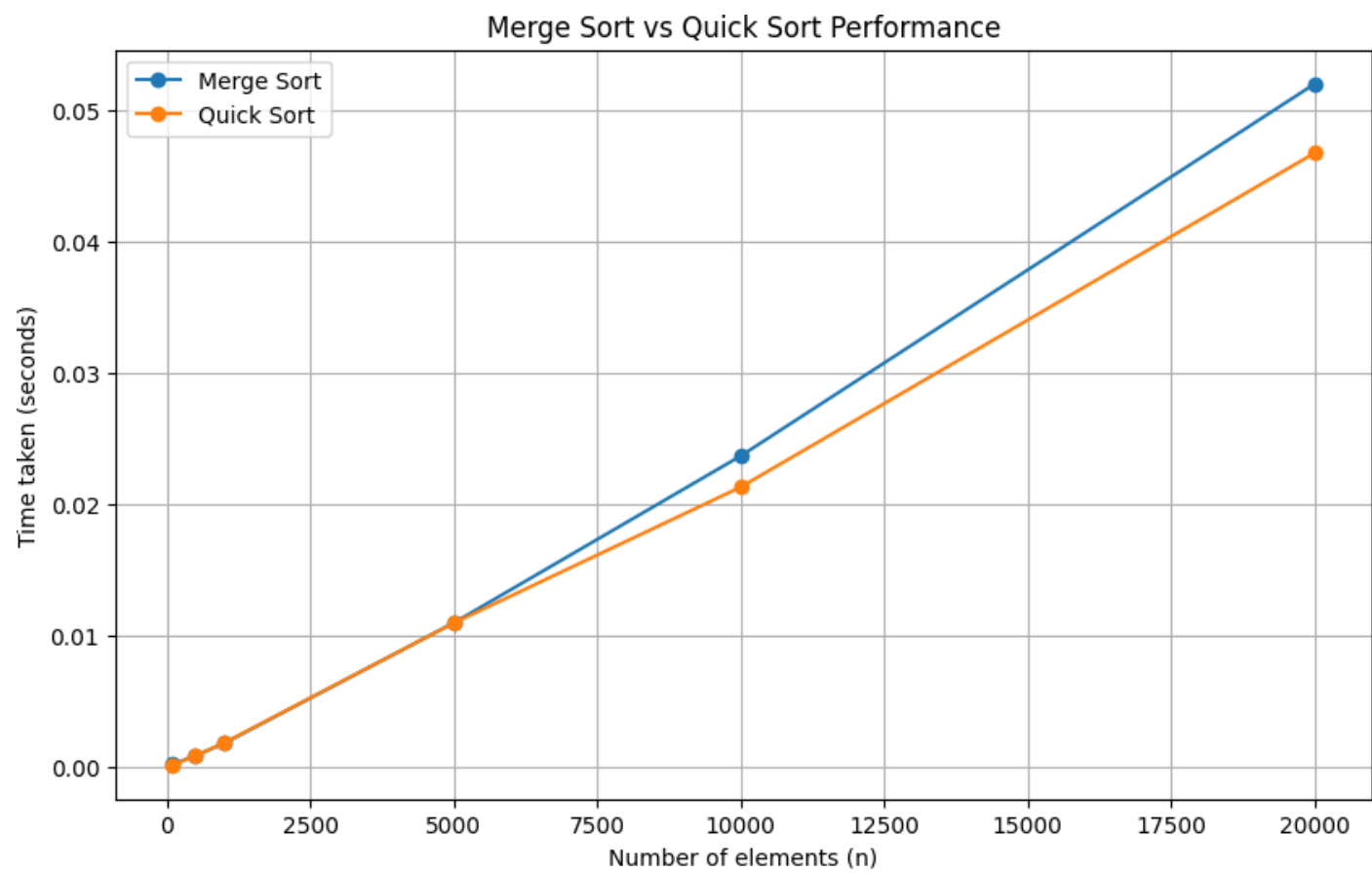
```
import random
import time
import matplotlib.pyplot as plt
# Merge Sort Implementation
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < 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
# Quick Sort Implementation
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 measure sorting times and plot the graph
def measure_and_plot():
    ns = [100, 500, 1000, 5000, 10000, 20000] # Different sizes of n
    merge_sort_times = []
    quick_sort_times = []
    for n in ns:
        # Generate a random list of size n
        data = [random.randint(1, 100000) for _ in range(n)]
        # Measure time for Merge Sort
        data_copy = data.copy()
        start_time = time.time()
        merge_sort(data_copy)
        merge_sort_times.append(time.time() - start_time)
        # Measure time for Quick Sort
        data_copy = data.copy()
        start_time = time.time()
        quick_sort(data_copy)
        quick_sort_times.append(time.time() - start_time)
    # Plot the results
    plt.figure(figsize=(10, 6))
    plt.plot(ns, merge_sort_times, label="Merge Sort", marker='o')
    plt.plot(ns, quick_sort_times, label="Quick Sort", marker='o')
    plt.xlabel("Number of elements (n)")
    plt.ylabel("Time taken (seconds)")
    plt.title("Merge Sort vs Quick Sort Performance")
    plt.legend()
    plt.grid()
    plt.show()
# Run the measurement and plotting
measure_and_plot()

```

OUTPUT:



RESULT:

Thus, Merge Sort algorithm is to efficiently sort an array or a list in ascending or descending order and Quick Sort algorithm is to efficiently sort an array or list of elements in ascending or descending order is executed successfully and its output get verified.

EX.NO:12	IMPLEMENT N QUEENS PROBLEM USING BACKTRACKING
DATE:	

AIM:

The N Queens problem is to place N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

ALGORITHM:

1. Start in the leftmost column.
2. If all queens are placed, return true.
3. Try all rows in the current column. For each row, check if the queen can be placed in that position without conflicting with already placed queens.
4. If a safe spot is found, mark this cell and recursively try to place queens in the next columns.
5. If placing queens in the current configuration leads to a solution, return true.
6. If placing queens in the current configuration does not lead to a solution, unmark this cell (backtrack) and try the next row.
7. If all rows have been tried and none worked, return false to trigger backtracking to the previous column.

PROGRAM:

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

def solve_n_queens_util(board, col, n):
    if col >= n:
        return True # All queens are placed successfully
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1 # Place queen
            if solve_n_queens_util(board, col + 1, n): # Recur to place queens in the next columns
                return True
            board[i][col] = 0 # If placing queen in the current configuration doesn't lead to a solution, backtrack
    return False

def solve_n_queens(n):
    # Initialize an empty chessboard
    board = [[0] * n for _ in range(n)]
```

```
if not solve_n_queens_util(board, 0, n):
    print("Solution does not exist")
    return
# Print the solution
print("N Queen problem")
print("Solution:")
for row in board:
    print(" ".join(map(str, row)))
# Example: Solve the 8-Queens problem
solve_n_queens(8)
```

OUTPUT:



N Queen problem

Solution:

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

RESULT:

Thus, N Queens problem is to place N chess queens on an $N \times N$ chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal is executed successfully and its output get verified.

EX.NO:13

RANDOMIZED ALGORITHM FOR TSP

DATE:

AIM:

The aim is to solve the Traveling Salesman Problem (TSP) using a randomized algorithm.

ALGORITHM:

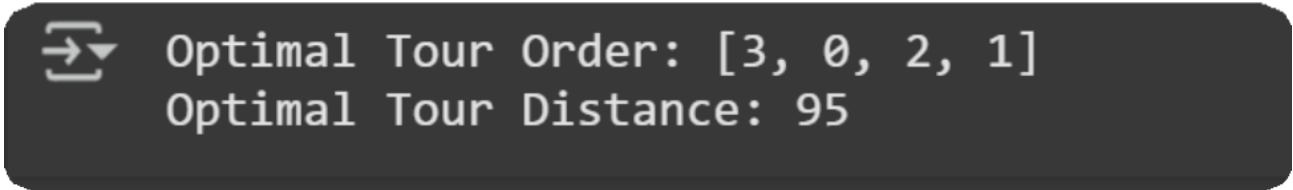
1. Randomized QuickSelect is an in-place algorithm that works by selecting a random pivot element from the list and partitioning the elements into two groups - those less than the pivot and those greater than the pivot.
2. The algorithm then recursively focuses on the group containing the desired k-th element until the element is found.

PROGRAM:

```
import random
import math
def calculate_total_distance(order, distance_matrix):
    total_distance = 0
    for i in range(len(order) - 1):
        total_distance += distance_matrix[order[i]][order[i + 1]]
    total_distance += distance_matrix[order[-1]][order[0]] # Return to the starting city
    return total_distance
def randomized_tsp(distance_matrix, max_iterations):
    num_cities = len(distance_matrix)
    current_solution = list(range(num_cities))
    current_distance = calculate_total_distance(current_solution, distance_matrix)
    for iteration in range(max_iterations):
        # Randomly choose two cities to swap
        city1, city2 = random.sample(range(num_cities), 2)
        # Swap the cities in the current solution
        current_solution[city1], current_solution[city2] = current_solution[city2], current_solution[city1]
        # Evaluate the total distance of the new solution
        new_distance = calculate_total_distance(current_solution, distance_matrix)
        # Accept the new solution if it is better or with a certain probability if it is worse
        if new_distance < current_distance or random.random() < math.exp((current_distance - new_distance) /
(iteration + 1)):
            current_distance = new_distance
    return current_solution, current_distance
# Example Usage:
# Replace distance_matrix with your actual distance matrix
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
```

```
max_iterations = 10000
result_order, result_distance = randomized_tsp(distance_matrix, max_iterations)
print("Optimal Tour Order:", result_order)
print("Optimal Tour Distance:", result_distance)
```

OUTPUT:

A dark-themed terminal window with a light gray icon of a terminal window on the left. The text inside the terminal is white and shows the output of the program.

```
⇒ Optimal Tour Order: [3, 0, 2, 1]
   Optimal Tour Distance: 95
```

RESULT:

Thus, the aim is to solve the Traveling Salesman Problem (TSP) using a randomized algorithm is executed successfully and its output get verified.

<u>EX.NO:14</u>	RANDOMIZED ALGORITHMS FOR FINDING THE KTH SMALLEST NUMBER
<u>DATE:</u>	

AIM:

The aim of Randomized Quick Select is to find the k-th smallest (or largest) element in an unordered list, similar to the Quick Sort algorithm. The key advantage of using Randomized Quick Select is its expected linear time complexity, making it efficient for finding the k-th order statistic.

ALGORITHM:

RANDOMIZED ALGORITHM FOR TSP:

1. **Initialization:**
 - Randomly shuffle the order of cities.
 - Initialize the current solution as the shuffled order.
2. **Main Loop:**
 - Repeat the following steps for a certain number of iterations or until convergence:
 - Randomly choose two cities in the current solution.
 - Swap the positions of the two chosen cities in the solution.
3. **Evaluation:**
 - Evaluate the total distance or cost of the new solution.
4. **Acceptance Criterion:**
 - If the new solution has a shorter total distance, accept it as the current solution.
 - If the new solution has a longer total distance, accept it with a certain probability, which decreases over time.
5. **Convergence:**
 - Repeat the main loop until the algorithm converges or reaches the maximum number of iterations.

PROGRAM:

```
import random
def partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    pivot = arr[high]
    i = low - 1
    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

def quickselect(arr, low, high, k):
    if low <= high:
        pivot_index = partition(arr, low, high)
        if pivot_index == k:
            return arr[pivot_index]
```

```
elif pivot_index < k:
    return quickselect(arr, pivot_index + 1, high, k)
else:
    return quickselect(arr, low, pivot_index - 1, k)

def kth_smallest(arr, k):
    if 0 < k <= len(arr):
        return quickselect(arr, 0, len(arr) - 1, k - 1)
    else:
        return "Invalid value of k"

# Example usage:
input_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
k_value = 4
result = kth_smallest(input_list, k_value)
print(f"The {k_value}th smallest element is: {result}")
```

OUTPUT:

The output is displayed in a dark-themed terminal window. On the left, there is a light blue icon of a terminal window with a cursor. To the right of the icon, the text "The 4th smallest element is: 3" is printed in a light blue monospaced font.

The 4th smallest element is: 3

RESULT:

Thus, the aim of Randomized Quick Select is to find the k-th smallest (or largest) element in an unordered list, similar to the Quick Sort algorithm is executed successfully and its output get verified..