## College Vision and Mission Statement

**Vision**

"Emerge as a premier technical institution of global standards, producing enterprising, knowledgeable engineers and entrepreneurs."

**Mission**

- Impart quality and contemporary technical education for rural students.
- Have the state of the art infrastructure and equipment for quality learning.
- Enable knowledge with ethics, values and social responsibilities.
- Inculcate innovation and creativity among students for contribution to society.

## Vision and Mission of the Department of Computer Science and Engineering

**Vision**

"Produce competent and intellectual computer science graduates by empowering them to compete globally towards professional excellence".

**Mission**

- Provide resources, environment and continuing learning processes for better exposure in latest and contemporary technologies in Computer Science and Engineering.
- Encourage creativity and innovation and the development of self-employment through knowledge and skills, for contribution to society
- Provide quality education in Computer Science and Engineering by creating a platform to enable coding, problem solving, design, development, testing and implementation of solutions for the benefit of society.

## Program Educational Objectives (PEOs)

The graduates of Computer Science and Engineering shall possess

**PEO I** **:** Have a successful career in computer software and hardware allied industries or shall pursue higher education or research or emerge as entrepreneurs.

**PEO II** **:** Have expertise in the areas of design and development of software and firmware solutions, real-time applications, web based solutions, etc.

**PEO III** **:** Contribute towards technological development through academic research and industrial practices and adapt to evolving technologies through life-long learning.

**PEO IV** **:** Practice their profession with good communication, leadership, ethics and social responsibility.

**PROGRAM OUTCOMES (POs)**:

**PO1: Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

**PO2: Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**PO3: Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**PO4: Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO5: Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**PO6: The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**PO7: Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO8: Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**PO9: Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**PO10: Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**PO12. Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Program Specific Outcomes (PSOs)**

**PSO1:** Exhibit design and programming skills to build and automate business solutions using cutting edge technologies.

**PSO2:** Strong theoretical foundation leading to excellence and excitement towards research, to provide elegant solutions to complex problems.

**PSO3:** Ability to work effectively with various engineering fields as a team to design, build and develop system applications.

V V COLLEGE OF ENGINEERING
(Approved By AICTE, New Delhi and Affiliated to Anna University, Chennai)
V V NAGAR, ARASOOR - 628656
Department of Computer Science and Engineering

# LABORATORY

## LIST OF EXPERIMENTS – R2021

| | |
|---|---|
| **PRACTICAL SUBJECT NAME** | ALGORITHMS LABORATORY |
| **PRACTICAL SUBJECT CODE** | CS3401 |
| **SEMESTER/ YEAR** | 04 / SECOND |
| **TOTAL HOURS** | 30 |
| **STAFF IN-CHARGE** | Dr.G.SUMILDA MERLIN |
| **LAB INSTRUCTOR** | Mrs. ANITHA |
| **REGULATION** | 2021 |

| | |
|---|---|
| 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 |

| S.No | Name of the Experiment | CO Mapping | PO Mapping |
|---|---|---|---|
| 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-PO4, PO9-PO12, PSO1-PSO3 |
| 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-PO4, PO9-PO12, PSO1-PSO3 |
| 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-PO4, PO9-PO12, PSO1-PSO3 |
| 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*. | CO1 | PO1-PO4, PO9-PO12, PSO1-PSO3 |
| 5 | Develop a program to implement graph traversal using Breadth First Search | CO2 | PO1-PO5, PO9-PO12, PSO1-PSO3 |
| 6 | Develop a program to implement graph traversal using Depth First Search | CO2 | PO1-PO5, PO9-PO12, PSO1-PSO3 |
| 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-PO5, PO9-PO12, PSO1-PSO3 |
| 8 | Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm. | CO2 | PO1-PO5, PO9-PO12, PSO1-PSO3 |
| 9 | Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem. | CO2 | PO1-PO5, PO9-PO12, PSO1-PSO3 |
| 10 | Compute the transitive closure of a given directed graph using Warshall's algorithm. | CO2 | PO1-PO5, PO9-PO12, PSO1-PSO3 |
| 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-PO5, PO9-PO12, PSO1-PSO3 |

| | | | |
|---|---|---|---|
| 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-PO5, PO9-PO12, PSO1-PSO3 |
| 13 | Implement N Queens problem using Backtracking. | CO4 | PO1-PO4, PO9-PO12, PSO1-PSO3 |
| 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-PO5, PO9-PO12, PSO1-PSO3 |
| 15 | Implement randomized algorithms for finding the $k^{th}$ smallest number. | CO5 | PO1-PO5, PO9-PO12, PSO1-PSO3 |

## RUBRICS FOR ASSESSING LABORATORY

| SI. No. | Criteria | Total Marks | Excellent (25) | Good (20) | Average (10) | Poor (5) |
|---|---|---|---|---|---|---|
| | | | 91% - 100% | 71% - 90% | 50% - 70% | <50% |
| 1 | Preparation | 25 | Gives clear idea about the aim and having good capability of executing experiments. | Capability of executing experiments but no proper clarification about the objective. | Gives clear idea about the target and has less capability of executing experiments. | Gives indistinct idea about the target and has less capability of executing experiments & who feel difficult to follow the objectives. |
| 2 | Viva | 25 | Have executed the experiments in an efficient way & make credible and unbiased judgments regarding the experiments. | Executed the experiments with less efficient & has partial judgments regarding the experiments. | Executed the experiments with less efficiency and has no judgments regarding experiments. | Incomplete experiments & lack of judgments regarding experiments. |
| 3 | Performanc e | 25 | Followed all the instructions given in the procedure and submitted the manual on time. | Followed all the instructions given in the procedure with some assisting. | Followed some of the instructions given in the procedure & late in submission of manual. | Unable to follow the instructions given in the procedure & late in submission of manual. |

| Department of Computer Science and Engineering | | |
|---|---|---|
| Preparation | 25 | |
| Viva | 25 | |
| Performance | 25 | |
| Total | 75 | |
| Lab Incharge | Date | |

| S.NO | DATE | NAME OF THE EXPERIMENT | SIGN |
|---|---|---|---|
| 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*. | |
| 5 | | Develop a program to implement graph traversal using Breadth First Search | |
| 6 | | Develop a program to implement graph traversal using Depth First Search | |
| 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. | |
| 8 | | Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm. | |
| 9 | | Implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem. | |
| 10 | | Compute the transitive closure of a given directed graph using Warshall's algorithm. | |
| 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. | |
| 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*. | |
| 13 | | Implement N Queens problem using Backtracking. | |
| 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. | |
| 14 | | Implement randomized algorithms for finding the $k^{th}$ smallest number. | |

## EX.NO. 1 Implement Linear Search. Determine the time required to search foran 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: The aim of this program is to perform linear search on an array of integers and measure the execution time

Algorithm:

Start the program.

1.Take the number of executions (T) as input.

2.Initialize an empty list for storing execution times (ot).

3.Initialize a variable temp to 0 for counting the number of times an element is found.

4.Initialize an empty list for storing input sizes (y).

5.For each execution j from 1 to T, do the following:

a. Take the size of the array (n) as input and append it to the list y.

b. Initialize an empty list my_list for storing the elements of the array.

c. Take n integers as input and append them to the list my_list.

d. Take the element to be searched (elem_to_search) as input.

e. Call the linearSearch function to search for the element in the array and measure the execution time.

f. If the element is found, print the index of the element, else print "Element not found!" and increment the temp vag. Append the execution time (tn) to the list ot.

6.Print the execution times for all the iterations.

7.Plot a graph of the execution time (X-axis) against the input size (Y-axis).

8.End the program.

**Ex.No.1. Implement Linear Search. Determine the time required to search foran 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.**

**Program:**

```
def linearSearch(array, n, x):
    # Going through array sequentially
    for i in range(0, n): if
        (array[i] == x):
            return i
    return -1

import matplotlib.pyplot as plt
import time
ot=[]
T=int(input("Enter no of Excution"))
temp=0
y=[]
for j in range(0,T):
    n=int(input("Enter n"))
    y.append(n) my_list=[]
    for i in range(0,n):
        ele=int(input("Enter Element"))
        my_list.append(ele)
    elem_to_search = int(input("Enter key"))
    print("The list is",my_list)
    start=time.time()
    my_result = linearSearch(my_list,n,elem_to_search)
    end=time.time()
    tn=end-start
    if my_result != -1:
        print("Element found at index ", str(my_result))
        else:
        print("Element not found!")
    temp=temp+1
    ot.append(tn)
print("Execution Time",ot)
# Plot a graph
X=ot
Y=y
plt.plot(X,Y)
plt.show()
```

## Ex.No.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: The aim of this program is to perform binary search on a sorted array of integers and measure the execution

Algorithm:

Start the program.

1.Take the number of executions (T) as input.

2.Initialize an empty list for storing execution times (ot).

3.Initialize an empty list for storing input sizes (yvalue).

4.For each execution j from 1 to T, do the following:

a. Take the size of the array (n) as input and append it to the list yvalue.

b. Initialize an empty list my_list for storing the elements of the array.

c. Take n integers as input and append them to the list my_list.

d. Sort the list my_list in ascending order.

e. Take the element to be searched (elem_to_search) as input.

f. Call the binary_search function to search for the element in the array and measure the execution time.

g. If the element is found, print the index of the element, else print "Element not found!".

h. Append the execution time (tn) to the list ot.

5.Print the execution times for all the iterations.

6.Plot a graph of the execution time (X-axis) against the input size (Y-axis).

7.End the program.

**Ex.No.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.**

**Program:**

```
# Python 3 program for recursive binary search.
# Modifications needed for the older Python 2 are found in comments.# Returns
index of x in arr if present, else -1

def binary_search(my_list, low, high, elem):
# Check base case
if high >= low:
        mid = (high + low) // 2
        # If element is present at the middle itself
        if my_list[mid] == elem:
                return mid
        # If element is smaller than mid, then it can only be present in#left subarray
        elif my_list[mid] > elem:
                return binary_search(my_list, low, mid - 1, elem)
        # Else the element can only be present in right subarray
        else:
                return binary_search(my_list, mid + 1, high, elem)
  else:
        # Element is not present in the array
        return -1

# Test array
import matplotlib.pyplot as plt
import time
ot=[]
yvalue=[]
T=int(input("Enter no of Excution"))
for j in range(0,T):
   n=int(input("Enter n"))
   yvalue.append(n)
   my_list=[]
   for i in range(0,n):
      ele=int(input("Enter Element"))
      my_list.append(ele)
   elem_to_search = int(input("Enter key"))
   print("The list is")
   for i in range(0,n):
      print(my_list[i])
   start=time.time()
```

```
    my_result = binary_search(my_list,0,len(my_list)-1,elem_to_search)
    end=time.time()
    tn=end-start
    if my_result != -1:
        print("Element found at index ", str(my_result))
    else:
        print("Element not found!")
    ot.append(tn)
print(“Execution time”,ot)
# Plot a graph
x=ot
y=yvalue
plt.plot(x,y)
plt.show()
```

**EX.NO.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**:

The aim of this program is to perform a pattern search in a given text and print the index of the starting position of the pattern in the text.

**Algorithm:**

Start the program.

1.Take the number of texts (tn) as input.

2.Initialize an empty list for storing the texts (txt).

3.For each text i from 1 to tn, do the following:

a. Take the text element as input and append it to the list txt.

4.Take the number of patterns (pn) as input.

5.Initialize an empty list for storing the patterns (pat).

6.For each pattern j from 1 to pn, do the following:

a. Take the pattern element as input and append it to the list pat.

7.Define a function called search that takes two parameters pat and txt.

8.Inside the search function, do the following:

a. Get the lengths of the pattern (M) and the text (N).

b. Iterate through each index i in the range (N - M + 1):

i. Set a counter variable j to zero.

ii. While j is less than M, compare the jth element of txt starting at the ith position

with the jth element of pat:

1. If they are not equal, break out of the loop.

2. If they are equal, increment j by 1.

iii. If j becomes equal to M, print "Pattern found at index i".

9.Call the search function for each pattern pat and text txt.

10.End the program.

**Ex. No. 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.**

**Program:**

```
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    for i in range(N - M + 1):
        j = 0
        while(j < M):
            if (txt[i + j] != pat[j]):
                break
            j += 1
        if (j == M):
            print("Pattern found at index ", i)
 # Driver's Code
tn=int(input("Enter no. of text"))
txt=[]
for i in range(0,tn):
    ele=str(input("Enter Text Elements"))
    txt.append(ele)
pat=[]
pn=int(input("Enter no. of pattern"))
for j in range(0,pn):
    el=str(input("Enter pattern Elements"))
    pat.append(el)
# Function call
search(pat, txt)
```

EX.NO.4a Sort a given set of elements using the Insertion sort methods and determine the time required to sort the elements. Repeat the experiment for different values of n, number of elements in the list to be sorted and plot a graph of the time take versus

**Aim:**

The aim of this program is to implement the insertion sort algorithm and measure the execution me for sorting an array of integers of different sizes. Additionally, the program plots a graph of execution time against the size of the input array.

**Algorithm:**

1.Define the insertion_sort function that takes an array as input.

2.Traverse through the array from index 1 to n-1.

3.Set the current element as the key.

4.Compare the key with the elements before it.

5.If the key is smaller than the jth element, move the jth element one position ahead.

6.Decrement j and repeat step 5 until the key is greater than or equal to the jth element.

7.Insert the key at the j+1th position.

8.In the main program, take the number of executions as input from the user.

9.For each execution, take the size of the input array as input from the user and then take

the array elements as input.

10.Call the insertion_sort function on the input array and measure the execution time using the

time module.

11.Store the execution time in a list for each execution.

12.Plot a graph of execution time against the size of the input array using the matplotlib module.

**Ex. No.4 a Sort a given set of elements using the Insertion 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 take versus n.**

**Program:**

```
def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]j
        = i - 1
        # Move elements of arr[0..i-1], that are greater than key, to one position ahead of their
 current position
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
#Function Call
 import matplotlib.pyplot as plt
 import time
ot=[]
 T=int(input("Enter no of Excution"))
 temp=0
x=[]
 for j in range(0,T):
    n=int(input("Enter n"))
    x.append(n) my_list=[]
    for i in range(0,n):
        ele=int(input("Enter Element"))
        my_list.append(ele)
    start=time.time()
    my_result = insertion_sort(my_list)
    end=time.time()
    time_comp=end-start
    print("Sorted array is:", my_list)
    temp=temp+1
ot.append(time_comp)
print("Execution Time",ot)
# Plot a graph
X=x
Y=ot
plt.plot(X,Y)
plt.show()
```

**EX.NO.4b Sort a given set of elements using the 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:

The aim of this program is to implement the heap sort algorithm to sort an array of integers in

ascending or descending order and to visualize the time taken to execute the sorting algorithm

for different input sizes.

**Algorithm:**

1. Define a function heapify() that takes three parameters: arr, n, and i.

2. Set largest to i (initialize largest as root).

3. Compute the index of the left child of the root as l=2i+1 and the index of the right child as r=2i+2.

4. If the left child exists and is greater than the root, set largest to l.

5. If the right child exists and is greater than the root, set largest to r.

6. If the largest value is not the root value, swap the largest and root values using a temporary variable.

7.Recursively call heapify() on the subtree with the largest value until the entire heap is sorted.

8. Define the main function heapSort() that takes a list arr as input.

9. Compute the length of the list as n = len(arr).

10. Build a max heap using heapify() by iterating over the array starting from the

last non-leaf node to the root.

11. One by one, extract elements from the heap and swap the root with the last element of the heap.

12. Heapify the new root to maintain the heap property.

13. The sorted array is obtained when all the elements have been extracted from the heap.

14. Define a function to plot a graph of input size vs. execution time.

15. Take input from the user for the number of executions T and the size of the array n for each execution.

16. For each execution, take input from the user for the n integers to be sorted.

17. Record the execution time for each execution in a list ot.

18. Plot the graph of input size vs. execution time using the matplotlib library.

19. Stop the pgm

**Ex. No. 4. b Sort a given set of elements using the 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**

**Program:**

```
# Python program for implementation of heap Sort
# To heapify subtree rooted at index i.
# n is size of heap

def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1 # left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2

  # See if left child of root exists and isgreater than root
    if l < n and arr[i] < arr[l]:
        largest = l
  # See if right child of root exists and is greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

  # Change root, if needed
  if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i]) # swap

   # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

  # Build a maxheap.
  # Since last parent will be at ((n//2)-1) we can start at that location.
  for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

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

# Driver code to test above
import matplotlib.pyplot as plt
import time
ot=[]
T=int(input("Enter no of Excution"))
temp=0
```

```
x=[]
for j in range(0,T):
n=int(input("Enter n"))
x.append(n)
my_list=[]
    for i in range(0,n):
        ele=int(input("Enter Element"))
        my_list.append(ele)
    start=time.time()
    my_result = heapSort(my_list)
    end=time.time()
    time_comp=end-start
    print("Sorted array is:", my_list)
    temp=temp+1
    ot.append(time_comp)
 print("Execution Time",ot)
 # Plot a graph
 X=x
 Y=ot
 plt.plot(X,Y)
 plt.show()
```

# Ex. No. 5 Develop a program to implement graph traversal using Breadth First Search

**Aim:**

The aim of this program is to implement the Breadth First Search (BFS) algorithm for traversing a graph

**Algorithm:**

1.Create a graph representation using adjacency list or adjacency matrix.

2.Create an empty list called 'visited' to keep track of all the visited nodes and an empty queue called 'queue' for the BFS traversal.

3.Choose a starting node and add it to the visited list and enqueue it in the queue.

4.While the queue is not empty, dequeue the node from the queue and print it.

5.For each adjacent node of the dequeued node, if it has not been visited before,mark it as visited and enqueue it into the queue.

6.Repeat steps 4-5 until the queue becomes empty.

7.Finally, the BFS traversal is complete.

**Ex. No. 5 Develop a program to implement graph traversal using Breadth First Search**

**Program:**

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

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

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

**Ex. No. 6 Develop a program to implement graph traversal using Depth First Search**

Aim: To implement Depth-First Search (DFS) algorithm on a graph represented as an adjacency list using graph

Algorithm:

1. Create a function named dfs that takes three parameters - visited, graph, and node.

2. Initialize an empty set named visited to keep track of visited nodes.

3. Check if the node is not in visited set, print the node, and add it to the visited set.

4. Loop through all the neighbors of the current node using a for loop.

5. Recursively call the dfs function for each neighbor of the current node that is not already visited.

6. Create a dictionary named graph that represents the adjacency list of the graph.

7. Initialize an empty set named visited.

8. Call the dfs function with initial parameters visited, graph, and a starting node.

9. Stop the pgm

**Ex. No. 6 Develop a program to implement graph traversal using Depth First Search**

**Program:**

```
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

## Ex. No. 7 Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

Aim: To implement Prim's Algorithm in Python for finding minimum spanning tree of a weighted graph.

Algorithm:

1. Initialize all nodes as unselected.

2. Choose any node to start with and mark it as selected.

3. Repeat until all nodes are selected:

 a. Find the minimum edge-weight from the selected nodes to the unselected nodes.

 b. Mark the corresponding node as selected.

 c. Add the selected edge to the minimum spanning tree.

4. Print the minimum spanning tree.

5. Stop the pgm

**Ex. No. 7 Find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.**

**Program:**

```
# Prim's Algorithm in Python

INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G = [[0, 19, 5, 0, 0],
     [19, 0, 5, 9, 2],
     [5, 5, 0, 1, 6],
     [0, 9, 1, 0, 1],
     [0, 2, 6, 1, 0]]

selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):

    minimum = INF
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```

## Ex.No.8: From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra's algorithm

**Aim**:

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

**Algorithm:**

Import the sys module.

**Define the graph's vertices and edges as lists.**

Define a function called "to_be_visited" that iterates over the vertices and returns the index of the vertex with the smallest distance that hasn't been visited yet.

Define the number of vertices and initialize a list called "visited_and_distance" to keep track of which vertices have been visited and their current distance from the source vertex.

Loop over each vertex in the graph and set its distance to infinity, except for the source vertex, whose distance is set to 0.

Define a loop that iterates over all vertices in the graph. In each iteration, find the next vertex to visit using the "to_be_visited" function.

Iterate over all the neighbors of the current vertex and update their distances if a shorter path is found.

Mark the current vertex as visited.

Loop over the "visited_and_distance" list and print out the shortest distance from the source vertex to each vertex in the graph.

```
# Dijkstra's Algorithm in Python

import sys

# Providing the graph

vertices = [[0, 0, 1, 1, 0, 0, 0],

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

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

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

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

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

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


edges = [[0, 0, 1, 2, 0, 0, 0],

        [0, 0, 2, 0, 0, 3, 0],

        [1, 2, 0, 1, 3, 0, 0],

        [2, 0, 1, 0, 0, 0, 1],

        [0, 0, 3, 0, 0, 2, 0],

        [0, 3, 0, 0, 2, 0, 1],

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


# Find which vertex is to be visited next

defto_be_visited():

globalvisited_and_distance

   v = -10

for index in range(num_of_vertices):
```

```
ifvisited_and_distance[index][0] == 0 \
and (v < 0 or visited_and_distance[index][1] <=
visited_and_distance[v][1]):
        v = index
return v


num_of_vertices = len(vertices[0])
visited_and_distance = [[0, 0]]
fori in range(num_of_vertices-1):
visited_and_distance.append([0, sys.maxsize])
for vertex in range(num_of_vertices):
   # Find next vertex to be visited
to_visit = to_be_visited()
forneighbor_index in range(num_of_vertices):
      # Updating new distances
if vertices[to_visit][neighbor_index] == 1 and \
visited_and_distance[neighbor_index][0] == 0:
new_distance = visited_and_distance[to_visit][1] \
            + edges[to_visit][neighbor_index]
ifvisited_and_distance[neighbor_index][1] >new_distance:
visited_and_distance[neighbor_index][1] = new_distance

visited_and_distance[to_visit][0] = 1
```

```
i = 0

# Printing the distance

for distance in visited_and_distance:

print("Distance of ", chr(ord('a') + i),

        " from source vertex: ", distance[1])

i = i + 1
```

**Ex.No.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

**Algorithm:**

1)Initialize the number of vertices as n, which is the length of the graph.

2)Create a dist matrix of size n x n and initialize all the values to infinity.

3)Copy the values of the input graph to the dist matrix.

4)Iterate through all the vertices, and for each vertex k, update the dist matrix by checking if the shortest path between i and j passes through k.

5)Return the dist matrix, which contains the shortest path between every pair of vertices.

**Program:**

```
# Floyd Warshall Algorithm in python

def floyd(graph):
# The number of vertices
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    # Initialize the distance matrix with the graph values
    for i in range(n):
        for j in range(n):
            dist[i][j] = graph[i][j]

    # Calculate shortest path for all vertices
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist

#input
graph= [[0, 3, INF, 5],[2, 0, INF, 4], [INF, 1, 0, INF], [INF, INF, 2, 0]]


floyd(graph)
```

**Ex.No.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

**Algorithm:**

Initialize the number of vertices as n, which is the length of the graph.

Create a closure matrix of size n x n and initialize all the values to 0.

Copy the values of the input graph to the closure matrix.

Iterate through all the vertices, and for each vertex k, update the closure matrix by checking if there exists a path from i to j through k.

Return the closure matrix, which contains the information about the existence of a path between every pair of vertices.

**Program:**

```
def transitive_closure(graph):
# The number of vertices
    n = len(graph)
    closure = [[0 for j in range(n)] for i in range(n)]

 # Initialize the distance matrix with the graph values
    for i in range(n):
        for j in range(n):
            closure[i][j] = graph[i][j]

    # Calculate shortest path for all vertices
    for k in range(n):
        for i in range(n):
            for j in range(n):
                closure[i][j] = closure[i][j] or (closure[i][k] and closure[k][j])

    return closure

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

closure = transitive_closure(graph)
print(closure)
```

**Ex.No.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 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: Define a function find_min_max(arr) that takes an array arr as input.

Step 2: Compute the length of the array n using the built-in len() function.

Step 3: Implement the base case for the recursion. If n equals 1, return the first element of the array as both the minimum and maximum.

Step 4: Implement the recursive case for when n is greater than 1 but less than or equal to 2. In this case, compare the two elements of the array and return them as the minimum and maximum accordingly.

Step 5: If n is greater than 2, divide the array into two halves by finding the middle index mid as n // 2.

Step 6: Recursively find the minimum and maximum of the left half of the array by calling find_min_max() on arr[:mid]. Store the results in left_min and left_max.

Step 7: Recursively find the minimum and maximum of the right half of the array by calling find_min_max() on arr[mid:]. Store the results in right_min and right_max.

Step 8: Combine the results from the left and right halves of the array to obtain the minimum and maximum of the entire array. The minimum is the smaller of the left_min and right_min, and the maximum is the larger of the left_max and right_max. Return these values as a tuple.

Step 9: Test the function by calling it on a sample input array and printing the result

**Program:**

```
def find_min_max(arr):
    n = len(arr)

    # Base case for recursion
    if n == 1:
        return arr[0], arr[0]

    # Recursive case
    elif n == 2:
        return (arr[0], arr[1]) if arr[0] < arr[1] else (arr[1], arr[0])

    else:
        mid = n // 2
        left_min, left_max = find_min_max(arr[:mid])
        right_min, right_max = find_min_max(arr[mid:])

        return min(left_min, right_min), max(left_max, right_max)

#input
arr=[2,7,3,6,5,1,9,4,8]
find_min_max(arr)
```

Ex.No. 12 a. **Implement Merge sort method to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.**

**Aim:**
To Implement Merge sort method to sort an array of elements and determine the time required to sort

**Algorithm**:
Step 1: Define a function named "merge" that takes an array "arr", and three integer parameters "l", "m", and "r". This function is used to merge two sorted subarrays into one sorted array.

Step 2: Calculate the size of the left and right subarrays using the given formulas. Then, create two temporary arrays L and R with size n1 and n2 respectively.

Step 3: Copy data from the original array "arr" to the temporary arrays L and R.

Step 4: Merge the two subarrays back into the original array "arr" using a while loop. Inside the loop, compare the first elements of the left and right subarrays, and copy the smaller element into the original array. Then, move the corresponding pointer forward in the respective

Step 5: Copy the remaining elements of either the left or right subarray

Program

```
# Python program for implementation of MergeSort
# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]

def merge(arr, l, m, r):

    n1 = m - l + 1
    n2 = r - m
     # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)
     # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]
    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
     # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
 # l is for left index and r is right index of the
# sub-array of arr to be sorted
```

```
def mergeSort(arr, l, r):
    if l < r:
         # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2
         # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
 # Driver code to test above
import matplotlib.pyplot as plt
import time
ot=[]
T=int(input("Enter no of Excution"))
temp=0
y=[]
for j in range(0,T):
  n=int(input("Enter n"))
  y.append(n)
  arr=[]
  for i in range(0,n):
    ele=int(input("Enter Element"))
    arr.append(ele)
  print("The list is",arr)
  start=time.time()
  mergeSort(arr, 0, n-1)
  end=time.time()
  tn=end-start
  print("\nSorted array is")
  for i in range(n):
    print("%d" % arr[i],end=" ")
  temp=temp+1
  ot.append(tn)
  print("\nExecution Time",ot)
# Plot a graph
X=y
Y=ot
plt.plot(X,Y)
plt.show()
```

**Ex.No. 12.b Implement Quick sort method to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.**

**AIM:**

To Implement Quick sort method to sort an array of elements and determine the time required to sort.

**ALGORITHM:**

1)Define a function named partition that takes an array, a low index, and a high index as parameters.

2)Inside the partition function, choose the rightmost element as the pivot.

3)Set a pointer i for the greater element, initialized to low - 1.

4)Traverse through all elements from low to high, compare each element with the pivot, and if an element smaller than or equal to the pivot is found, swap it with the greater element pointed by i and increment i.

5)Swap the pivot element with the greater element specified by i+1.

6)Return the position from where partition is done, i+1.

7)Define a function named quickSort that takes an array, a low index, and a high index as parameters.

8)Inside the quickSort function, find the pivot element such that elements smaller than pivot are on the left and elements greater than pivot are on the right. Then, make two recursive calls to quickSort function, one on the left of the pivot and another on the right of the pivot.

**Program:**

```
# Python program for implementation of Quicksort Sort
# This implementation utilizes pivot as the last element in the nums li
st
# It has a pointer to keep track of the elements smaller than the pivot
# At the very end of partition() function, the pointer is swapped with
the pivot
# to come up with a "sorted" nums relative to the pivot
# Function to find the partition position
def partition(array, low, high):
     # choose the rightmost element as pivot
    pivot = array[high]
    # pointer for greater element
    i = low - 1
    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:
            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1
             # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])
     # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
     # Return the position from where partition is done
    return i + 1
# function to perform quicksort
def quickSort(array, low, high):
    if low < high:
         # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)
         # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)
        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)

import matplotlib.pyplot as plt
import time
ot=[]
T=int(input("Enter no of Excution"))
temp=0
```

```
y=[]
for j in range(0,T):
  n=int(input("Enter n"))
  y.append(n)
  arr=[]
  for i in range(0,n):
    ele=int(input("Enter Element"))
    arr.append(ele)
  print("The list is",arr)
  start=time.time()
  quickSort(arr, 0, n - 1)
  end=time.time()
  tn=end-start
  print('Sorted Array in Ascending Order:')
  print(arr)
  temp=temp+1
  ot.append(tn)
  print("\nExecution Time",ot)
# Plot a graph 4
X=y
Y=ot
plt.plot(X,Y)
plt.show()
```

**EX.No.13** Implement N Queens problem using Backtracking.

**Aim:** To implement the N Queens problem using Backtracking approach

**Algorithm**

Step 1. Start in the leftmost column
Step 2. If all queens are placed return true
Step 3. Try all rows in the current column. Do following for every tried row.
a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
b) If placing queen in [row, column] leads to a solution then return true.
c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
Step 4. If all rows have been tried and nothing worked, return false to trigger backtracking.

**Program:**

```
# Python3 program to solve N Queen Problem using backtracking

global N
N = 4
def printSolution(board):
    for i in range(N):
        for j in range(N):
            if board[i][j] == 1:
                print("Q",end=" ")
            else:
                print(".",end=" ")
        print()

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

def solveNQUtil(board, col):
    # Base case: If all queens are placed then return true
    if col >= N:
        return True
```

```
    #Consider this column & try placing this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1
            # Recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True
            # If placing queen in board[i][col]
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0
     # If the queen cannot be placed in any row in
    # this column col then return false
    return False

def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]

    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True

# Driver Code
if __name__ == '__main__':
    solveNQ()
```

**Result:**

Thus the N Queens problem using Backtracking approach was developed successfully.

**Ex.No. 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 implement any scheme to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm.

**Algorithm:**

Step 1: Start on an arbitrary vertex as current vertex.
Step 2: Find out the shortest edge connecting current vertex and an unvisited vertex V.
Step 3: Set current vertex to V.
Step 4: Mark V as visited.
Step 5: If all the vertices in domain are visited, then terminate.
Step 6: Go to step 2.
Step 7: The sequence of the visited vertices is the output of the algorithm.

**Program:**

```
from sys import maxsize
from itertools import permutations
V = 4
# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        # store current Path weight(cost)
        current_pathweight = 0
        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        # update minimum
        min_path = min(min_path, current_pathweight)
    return min_path
# Driver Code
```

```
if __name__ == "__main__":
    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],[15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

**VIVA QUESTIONS:**

1. Define Optimal Solution.
2. Explain Travelling Sales Person Problem.
3. What is the time complexity of Travelling Sales Person Problem?

**Result:**

Thus the optimal solution was obtained successfully for the Traveling Salesperson problem.

**Ex.No 15** Implement randomized algorithms for finding the $k^{th}$ smallest number.

**Aim:**

To implement randomized algorithms for finding the $k^{th}$ smallest number.

**Algorithm:**

Step 1: start with pushing all the array elements in the set
Step 2: then traverse through the set and return its k-1th element
Step 3: which will be the $k^{th}$ smallest element of the array.

**Program:**

```
def kth_smallest_el(lst, k):
   lst.sort()
   return lst[k-1]
nums = [1,2,4,3,5,4,6,9,2,1]
print("Original list:")
print(nums)
k = 1
for i in range(1, 11):
   print("kth smallest element in the said list, when k = ",k)
   print(kth_smallest_el(nums, k))
   k=k+1
```

**VIVA QUESTIONS:**

1. Can a min-heap be used to solve $K^{th}$ largest element?
2. What is the most efficient approach to solve the $K^{th}$ largest element?

**Result:**

Thus the randomized algorithms for finding the $k^{th}$ smallest number was successfully implemented.