EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

'Jnana Prabha', Virgo Nagar Post, Bengaluru-560049

Department of Computer Science and Engineering

Academic Year: 2023-24

LABORATORY MANUAL

Semester	**: IV**
Subject	**: Analysis and Design of Algorithms Laboratory**
Subject Code	**: BCDL404**

NAME: ______________________________

USN: ______________________________

SECTION: ______________________________

PROGRAM OUTCOMES

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 analyse 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 modernengineering and IT tools including prediction and modeling to complex engineering activitieswith 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.

EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

INSTITUTE VISION AND MISSION

VISION

The East Point College of Engineering and Technology aspires to be a globally acclaimed institution, recognized for excellence in engineering education, applied research and nurturing students for holistic development.

MISSION

M1: To create engineering graduates through quality education and to nurtureinnovation, creativity and excellence in teaching, learning and research

M2: To serve the technical, scientific, economic and societal developmental needsof our communities

M3: To induce integrity, teamwork, critical thinking, personality development andethics in students and to lay the foundation for lifelong learning

EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

DEPARTMENT VISION AND MISSION

VISION

The department aspires to be a Centre of excellence in Computer Science & Engineering to develop competent professionals through holistic development.

MISSION

M1: To create successful Computer Science Engineering graduates through effective pedagogies, the latest tools and technologies, and excellence in teaching and learning.

M2: To augment experiential learning skills to serve technical, scientific, economic, and social developmental needs.

M3: To instil integrity, critical thinking, personality development, and ethics in students for a successful career in Industries, Research, and Entrepreneurship.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO 1: To produce graduates who can perform technical roles to contribute effectively in software industries and R&D Centre.

PEO 2: To produce graduates having the ability to adapt and contribute in key domains of computer science and engineering to develop competent solutions.

PEO 3: To produce graduates who can provide socially and ethically responsible solutions while adapting to new trends in the domain to carve a successful career in the industry.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: To conceptualize, model, design, simulate, analyse, develop, test, and validate computing systems and solve technical problems arising in the field of computer science & engineering.

PSO2: To specialize in the sub-areas of computer science & engineering systems such as cloud computing, Robotic Process Automation, cyber security, big data analytics, user interface design, and IOT to meet industry requirements.

PSO3: To build innovative solutions to meet the demands of the industry using appropriate tools and techniques.

COURSE LEARNING OBJECTIVES

CLO1: To design and implement various algorithms in C/C++ programming using suitable development tools to address different computational challenges.

CLO2: To apply diverse design strategies for effective problem-solving.

CLO3: To Measure and compare the performance of different algorithms to determine their efficiency and suitability for specific tasks.

COURSE OUTCOMES

At the end of the course the student will be able to:

CO1: Develop programs to solve computational problems using suitable algorithm design strategy.

CO2: Compare algorithm design strategies by developing equivalent programs and observing running times for analysis (Empirical).

CO3: Make use of suitable integrated development tools to develop programs

CO4: Choose appropriate algorithm design techniques to develop solution to the computational and complex problems.

CO5: Demonstrate and present the development of program, its execution and running time(s) and record the results/inferences.

Analysis & Design of Algorithms Lab		**Semester**	**4**
Course Code	**BCDL404**	**CIE Marks**	**50**
Teaching Hours/Week (L:T:P: S)	**0:0:2:0**	**SEE Marks**	**50**
Credits	**01**	**Exam Hours**	**2**
Examination type (SEE)	**Practical**		

Sl. No	**Experiments**
1	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.
2	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.
3	Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm. Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.
4	Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.
5	Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.
6	Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.
7	Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.
8	Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.
9	Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
10	Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

11	Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
12	Design and implement C/C++ Program for N Queen's problem using Backtracking.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation (CIE):

CIE marks for the practical course are **50 Marks**.

The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

Each experiment is to be evaluated for conduction with an observation sheet and record write-up.

Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.

Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.

Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).

Weightage to be given for neatness and submission of record/write-up on time.

Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.

In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.

The suitable rubrics can be designed to evaluate each student’s performance and learning ability.

The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student.

Semester End Evaluation (SEE):

SEE marks for the practical course are 50 Marks.

SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute.

The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to be conducted between the schedule mentioned in the academic calendar of the University.

All laboratory experiments are to be included for practical examination.

(Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. **OR** based on the course requirement evaluation rubrics shall be decided jointly by examiners.

Students can pick one question (experiment) from the questions lot prepared by the examiners jointly.

Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners.

General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in -60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)

Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.

The minimum duration of SEE is 02 hours

Suggested Learning Resources:

Virtual Labs (CSE): http://cse01-iiith.vlabs.ac.in/

Index

Course Articulation Matrix

Cos	Pos												PSOs		
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12	PSO 1	PSO 2	PSO 3
CO1	3	3	3	-	3	2	-	-	3	-	-	1	3	2	3
CO2	3	3	3	-	3	2	-	-	3	-	-	1	3	2	3
CO3	3	3	3	-	3	2	-	-	3	-	-	1	3	2	3
CO4	3	3	3	-	3	2	-	-	3	-	-	1	3	2	3
CO5	3	3	3	-	3	2	-	-	3	-		1	3	2	3

3 - High Correlation 2 - Medium Correlation 1 – Low Correlation

INTRODUCTION TO ANALYSIS AND DESIGN OF ALGORITHMS

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

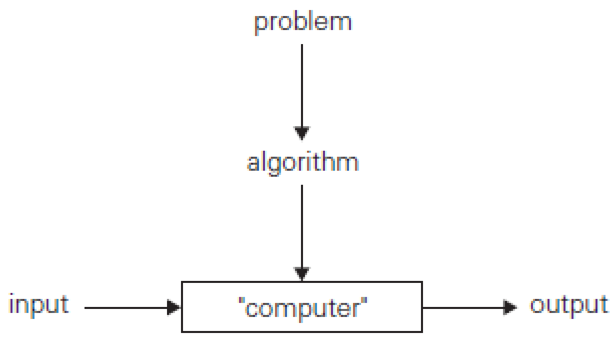


Figure: Notion of the Algorithm

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

Algorithms are at the heart of computer science, serving as the fundamental building blocks for solving a myriad of real-world problems. From optimizing resource allocation to routing data through networks, algorithms play a crucial role in virtually every aspect of computing. Understanding how to design efficient algorithms is therefore essential for any aspiring computer scientist or software engineer.

This laboratory course is designed to provide you with hands-on experience in analyzing, designing, and implementing algorithms using a variety of techniques and strategies. Through a series of programming exercises and practical assignments, you will learn to apply algorithmic principles to solve diverse computational problems, ranging from graph theory and network optimization to sorting and searching.

The most important problem types are:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

EXPERIMENT - 01

Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.

About the Program:

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$;   $ecounter \leftarrow 0$      //initialize the set of tree edges and its size
    $k \leftarrow 0$                          //initialize the number of processed edges
    while $ecounter < |V| - 1$ do
        $k \leftarrow k + 1$
        if $E_T \cup \{e_{i_k}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$;   $ecounter \leftarrow ecounter + 1$
    return $E_T$
```

Program:

```
#include<stdio.h>
#define MAX_VEREX 9
#define INFINITY 999
int parent[MAX_VER];

int find(int i)
{
    while (parent[i])
        i = parent[i];
    return i;
}
int uni(int i, int j)
{
    if (i != j)
 {
        parent[j] = i;
        return 1;
    }
    return 0;
}
void kruskal(int n, int cost[MAX_VERTEX][MAX_VERTEX]) {
    int ne = 1, mincost = 0, a, b, u, v, min;
    printf("The edges of Minimum Cost Spanning Tree are\n");
    while (ne < n) {
        min = INFINITY; // Reset min to infinity
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (cost[i][j] < min) {
                    min = cost[i][j];
                    a = u = i;
                    b = v = j;
                }
            }
        }
        u = find(u);
        v = find(v);
        if (uni(u, v)) {
            printf("%d edge (%d,%d) = %d\n", ne++, a, b, min);
            mincost += min;
        }
        cost[a][b] = cost[b][a] = INFINITY; // Update to use INFINITY for infinity
    }
    printf("\n\tMinimum cost = %d\n", mincost);
}

int main() {
    int n, cost[MAX_VERTEX][MAX_VERTEX];
    printf("\n\tImplementation of Kruskal's algorithm\n");
    printf("Enter the no. of vertices: ");
    scanf("%d", &n);
    printf("Enter the cost adjacency matrix:\n");
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INFINITY; // Update to use INFINITY for infinity
        }
        }
        kruskal(n, cost);
        return 0;
}
```

<u>Sample Output:</u>

```
Implementation of Kruskal's algorithm
Enter the no. of vertices:4
Enter the cost adjacency matrix:
0  3   2    1
3  0  999 2
2 999 0   4
1  2   4    0

The edges of Minimum Cost Spanning Tree are
1 edge (1,4) =1
2 edge (1,3) =2
3 edge (2,4) =2
Minimum cost = 5
```

OUTPUT:

EXPERIMENT -2

Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

About the Program:

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

a) First, we have to initialize an Minimum Spanning Tree with the randomly chosen vertex.

b) Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.

c) Repeat step 2 until the minimum spanning tree is formed.

The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Algorithm:

```
ALGORITHM  Prim(G)
    //Prim's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$   //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

Program:

```
#include<stdio.h>
#define MAX_NODES 10
#define INF 999
int n, ne = 1, mincost = 0;
int visited[MAX_NODES] = {0}, cost[MAX_NODES][MAX_NODES];

int main() {
   int a, b, u, v, min;
   printf("Implementation of Prim's Algorithm");
   printf("\nEnter the number of nodes: ");
   scanf("%d", &n);

   printf("\nEnter the adjacency matrix:\n");
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
            cost[i][j] = INF;
      }
   }

   visited[1] = 1;
```

```
    while (ne < n) {
        min = INF;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (cost[i][j] < min) {
                    if (visited[i] != 0) {
                        min = cost[i][j];
                        a = u = i;
                        b = v = j;
                    }
                }
            }
        }

        if (visited[u] == 0 || visited[v] == 0) {
            printf("\n Edge %d: (%d %d) cost: %d", ne++, a, b, min);
            mincost += min;
            visited[b] = 1;
        }
        cost[a][b] = cost[b][a] = INF;
    }
    printf("\n Minimum cost = %d\n", mincost);
    return 0;
}
```

Sample Output:

```
Implementation of Prim's Algorithm
Enter the number of nodes: 4

Enter the adjacency matrix:
0    3    2    1
3    0   999 2
2   999   0  4
1     2    4   0
```

```
Edge 1: (1 4) cost: 1
Edge 2: (1 3) cost: 2
Edge 3: (4 2) cost: 2
Minimum cost = 5
```

OUTPUT:

EXPERIMENT-3

a) **Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.**

b) **Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.**

About the Program:

a) Given a weighted connected graph (undirected or directed), the ***all-pairs shortest-paths problem*** asks to find the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.

```
ALGORITHM  Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
    $D \leftarrow W$  //is not necessary if W can be overwritten
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
    return D
```

Program:

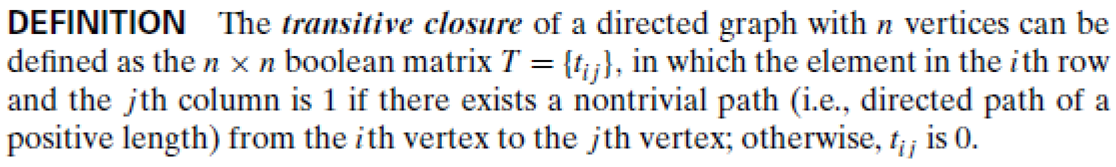
```
#include <stdio.h>
void floyd(int a[4][4], int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (a[i][j] > a[i][k] + a[k][j]) {
                    a[i][j] = a[i][k] + a[k][j];
                }
            }
        }
    }
    printf("All Pairs Shortest Path is :\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                printf("%d ", a[i][j]);
            }
            printf("\n");
        }
    }

    int main() {
        int n;
        int cost[n][n];
        printf("Enter the number of nodes: ");
        scanf("%d", &n);
        printf("Enter the cost matrix (%d x %d):\n", n, n);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &cost[i][j]);
            }
        }
        floyd(cost, n);
        return 0;
}
```

Sample Output:

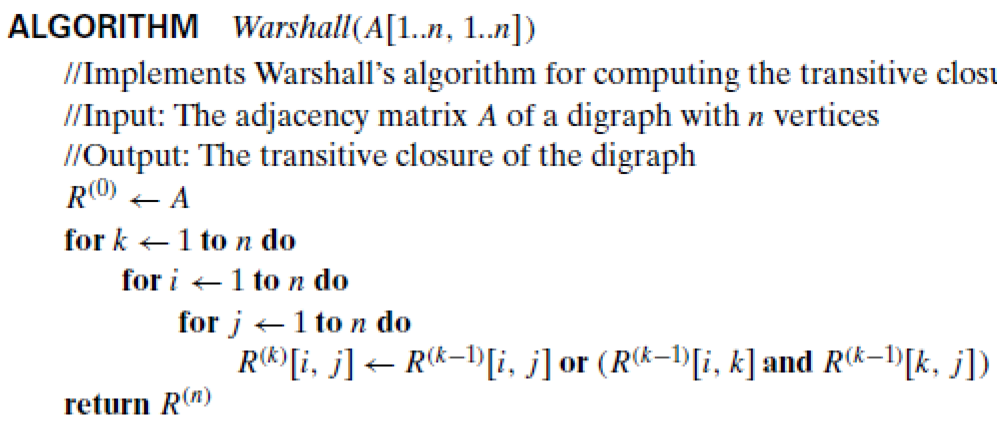
```
    Enter the number of nodes: 4
    Enter the cost matrix (4 x 4):
    0 3 999 4
    8 0 2 999
    5 999 0 1
    2 999 999 0
    All Pairs Shortest Path is :
    0 3 5 4
    5 0 2 3
    3 6 0 1
    2 5 7 0
```

OUTPUT:

b) **Warshall's algorithm constructs the transitive closure through a series of *n* × *n* boolean matrices.**



DEFINITION The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.



```
ALGORITHM  Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

<u>**Program:**</u>

```
#include <stdio.h>
#define MAX 20
```

```
void display(int matrix[MAX][MAX], int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%3d", matrix[i][j]);
        }
        printf("\n");
    }
}

void warshall(int adj[MAX][MAX], int n) {
    int path[MAX][MAX];
    int i, j, k;

    // Initialize the path matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            path[i][j] = adj[i][j];
        }
    }

    // Warshall's algorithm
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                path[i][j] = (path[i][j] || (path[i][k] && path[k][j]));
            }
        }
    }

    printf("Transitive closure matrix is:\n");
    display(path, n);
}
```

```
int main() {
    int adj[MAX][MAX];
    int n, i, j;

    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter the adjacency matrix (1 for connected, 0 for not connected):\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &adj[i][j]);
        }
    }
    printf("Given adjacency matrix is:\n");
    display(adj, n);
    warshall(adj, n);
    return 0;
}
```

Sample Output:

```
Enter the number of vertices: 4
Enter the adjacency matrix (1 for connected, 0 for not connected):
0 1 1 1
0 0 1 0
0 0 0 1
1 1 1 0
Given adjacency matrix is:
  0  1  1  1
  0  0  1  0
  0  0  0  1
  1  1  1  0
Transitive closure matrix is:
  1  1  1  1
  1  1  1  1
  1  1  1  1
  1  1  1  1
```

OUTPUT:

EXPERIMENT-4

Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.

About the Program:

Single-source shortest-paths problem: for a given vertex called the ***source*** in a weighted connected graph, find shortest paths to all its other vertices. The best-known algorithm for the single-source shortest-paths problem, called ***Dijkstra's algorithm***. This algorithm is applicable to undirected and directed graphs with nonnegative weights only. The obvious but probably most widely used applications are transportation planning and packet routing in communication networks, including the Internet.

Algorithm:

```
ALGORITHM  Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
    //          and its vertex $s$
    //Output: The length $d_v$ of a shortest path from $s$ to $v$
    //            and its penultimate vertex $p_v$ for every vertex $v$ in $V$
    Initialize(Q)    //initialize priority queue to empty
    for every vertex $v$ in $V$
        $d_v \leftarrow \infty$;   $p_v \leftarrow$ null
        Insert(Q, v, $d_v$)   //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$;   Decrease(Q, s, $d_s$)   //update priority of $s$ with $d_s$
    $V_T \leftarrow \varnothing$
    for $i \leftarrow 0$ to $|V| - 1$ do
        $u^* \leftarrow$ DeleteMin(Q)   //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) < d_u$
                $d_u \leftarrow d_{u^*} + w(u^*, u)$;   $p_u \leftarrow u^*$
                Decrease(Q, u, $d_u$)
```

Program:

```
#include<stdio.h>
#define INFINITY 9999
#define MAX 10
void dijkstra(int G[MAX][MAX], int n, int startnode);
void printPath(int pred[], int start, int end);
```

```
int main() {
    int G[MAX][MAX], n, u;
    printf("\nEnter the number of vertices: ");
    scanf("%d", &n);
    printf("\nEnter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &G[i][j]);
        }
    }
   printf("\nEnter the starting node: ");
   scanf("%d", &u);
    dijkstra(G, n, u);
    return 0;
}

void dijkstra(int G[MAX][MAX], int n, int startnode) {
    int distance[MAX], pred[MAX];
    int visited[MAX] = {0}, count, mindistance, nextnode;
    for (int i = 0; i < n; i++) {
        distance[i] = G[startnode][i];
        pred[i] = startnode;
    }
    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;
    while (count < n - 1) {
        mindistance = INFINITY;
        for (int i = 0; i < n; i++) {
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }
        }
```

```
            visited[nextnode] = 1;
            for (int i = 0; i < n; i++) {
                if (!visited[i] && mindistance + G[nextnode][i] < distance[i]) {
                    distance[i] = mindistance + G[nextnode][i];
                    pred[i] = nextnode;
                }
            }
            count++;
        }
        for (int i = 0; i < n; i++) {
            if (i != startnode) {
                printf("\nDistance of %d = %d", i, distance[i]);
                printf("\nPath = ");
                printPath(pred, startnode, i);
            }
        }
    }

    void printPath(int pred[], int start, int end) {
        if (end == start) {
            printf("%d", start);
            return;
        }
        printPath(pred, start, pred[end]);
        printf(" -> %d", end);
}
```

Sample Output:

```
Enter the number of vertices: 5
Enter the adjacency matrix:
0 1 1 4 999
999 0 999 999 999
999 999 0 2 999
999 999 999 0 3
999 999 999 999 0
```

```
Enter the starting node: 0
Distance of 1 = 1
Path = 0 -> 1
Distance of 2 = 1
Path = 0 -> 2
Distance of 3 = 3
Path = 0 -> 2 -> 3
Distance of 4 = 6
Path = 0 -> 2 -> 3 -> 4
```

OUTPUT:

EXPERIMENT-5

Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

About the Program:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a DAG, and topological sorting of its vertices is impossible.

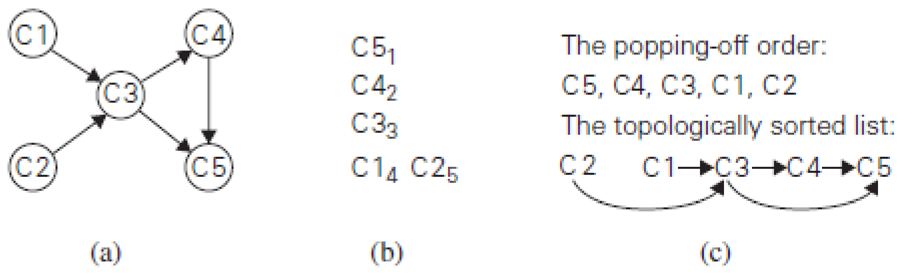


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Program:

```
#include <stdio.h>
#define MAX 10
void topologicalSort(int graph[MAX][MAX], int n);

int main(void) {
   int graph[MAX][MAX], n;
   printf("Topological Sorting Algorithm\n");
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   printf("Enter the adjacency matrix:\n");
```

```
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
    topologicalSort(graph, n);
    printf("\n");
    return 0;
}

void topologicalSort(int graph[MAX][MAX], int n) {
    int inDegree[MAX] = {0};
    int stack[MAX], top = -1;
    int sortedOrder[MAX], orderIndex = 0;
    // Calculate in-degree for each vertex
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (graph[j][i] == 1)
                inDegree[i]++;

    // Push vertices with in-degree 0 onto the stack
    for (int i = 0; i < n; i++)
        if (inDegree[i] == 0)
            stack[++top] = i;

    // Perform topological sorting
    while (top != -1) {
        int vertex = stack[top--];
        sortedOrder[orderIndex++] = vertex;

        for (int i = 0; i < n; i++) {
            if (graph[vertex][i] == 1) {
                inDegree[i]--;
                if (inDegree[i] == 0)
                    stack[++top] = i;
            }
            }
        }

        // Print sorted order
        printf("Topological Sorting (JOB SEQUENCE):\n");
        for (int i = 0; i < orderIndex; i++)
            printf("%d ", sortedOrder[i] + 1);
}
```

Sample Output:

```
Topological Sorting Algorithm -
Enter the number of vertices : 4
Enter the adjacency matrix -
0 1 1 0
0 0 0 1
0 0 0 0
0 0 0 0
Topological Sorting (JOB SEQUENCE) is:-
1 3 2 4
```

OUTPUT:

EXPERIMENT-6

Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.

About the Program:

We are given n items with some weights and corresponding values and a knapsack of capacity W. The items should be placed in the knapsack in such a way that the total value is maximum and total weight should be less than knapsack capacity.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Program:

```
#include<stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int W, int wt[], int val[], int n) {
    int K[n + 1][W + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
```

```
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}

int main() {
    int n, val[20], wt[20], W;

    printf("Enter number of items: ");
    scanf("%d", &n);

    printf("Enter value and weight of items:\n");
    for (int i = 0; i < n; ++i) {
        scanf("%d %d", &val[i], &wt[i]);
    }

    printf("Enter size of knapsack: ");
    scanf("%d", &W);

    printf("Maximum value: %d\n", knapsack(W, wt, val, n));
    return 0;
}
```

Sample Output:

```
Enter number of items: 4
Enter value and weight of items:
10     2
50     3
20     6
40     1
Enter size of knapsack: 5
```

```
Maximum value: 90
```

OUTPUT:

EXPERIMENT 7

Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

About the Program:

The fractional knapsack problem means that we can divide the item. For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

Program:

```
#include <stdio.h>
#define MAX_ITEMS 50

void solveDiscreteKnapsack(float weight[], float profit[], int n, float capacity) {
    float ratio[MAX_ITEMS], totalValue = 0;

    // Calculate profit-to-weight ratio for each item
    for (int i = 0; i < n; i++)
        ratio[i] = profit[i] / weight[i];

    // Sort items in non-increasing order of profit-to-weight ratio
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
                float temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }

    // Fill the knapsack with items
    for (int i = 0; i < n && capacity > 0; i++) {
        if (weight[i] <= capacity) {
            totalValue += profit[i];
            capacity -= weight[i];
        } else {
            totalValue += ratio[i] * capacity;
            break;
        }
    }

    printf("Maximum value for 0/1 knapsack problem: %.2f\n", totalValue);
}

void solveContinuousKnapsack(float weight[], float profit[], int n, float capacity) {
    float ratio[MAX_ITEMS], totalValue = 0;

    // Calculate profit-to-weight ratio for each item
    for (int i = 0; i < n; i++)
        ratio[i] = profit[i] / weight[i];

    // Sort items in non-increasing order of profit-to-weight ratio
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
                float temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;
                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }

    // Fill the knapsack with items
    for (int i = 0; i < n && capacity > 0; i++) {
        totalValue += profit[i];
        capacity -= weight[i];
    }

    printf("Maximum value for continuous knapsack problem: %.2f\n", totalValue);
}

int main() {
    int n;
    float capacity;

    printf("Enter the number of items: ");
    scanf("%d", &n);

    float weight[MAX_ITEMS], profit[MAX_ITEMS];
    printf("Enter the weight and profit of each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%f %f", &weight[i], &profit[i]);
    }
```

```
        printf("Enter the capacity of the knapsack: ");
        scanf("%f", &capacity);

        solveDiscreteKnapsack(weight, profit, n, capacity);
        solveContinuousKnapsack(weight, profit, n, capacity);

        return 0;
}
```

Sample Output:

```
Enter the number of items: 4
Enter the weight and profit of each item:
Item 1:     2     10
Item 2:     3     50
Item 3:     6     20
Item 4:     1     40
Enter the capacity of the knapsack: 5
Maximum value for 0/1 knapsack problem: 95.00
Maximum value for continuous knapsack problem: 100.00
```

OUTPUT:

EXPERIMENT -8

Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.

About the Program:

The ***subset-sum problem*** is to find a subset of a given set *A* = {*a*1, . . . , *an*} of *n* positive integers whose sum is equal to a given positive integer *d*. For example, for *A* = {1, 2, 5, 6, 8} and *d* = 9, there are two solutions: {1, 2, 6} and {1, 8}.

Program:

```
#include <stdio.h>
#define MAX_SIZE 10

int set[MAX_SIZE], subset[MAX_SIZE];
int n, targetSum, flag = 0;

void display(int count) {
    printf("{");
    for (int i = 0; i < count; i++) {
        printf("%d", subset[i]);
        if (i < count - 1) {
            printf(", ");
        }
    }
    printf("}\n");
}

void findSubset(int sum, int index, int count) {
    if (sum == targetSum) {
        flag = 1;
        display(count);
        return;
    }
```

```
        if (sum > targetSum || index >= n) {
            return;
        }
        subset[count] = set[index];
        findSubset(sum + set[index], index + 1, count + 1);
        findSubset(sum, index + 1, count);
    }

    int main() {
        printf("Enter the number of elements in the set: ");
        scanf("%d", &n);

        printf("Enter the elements of the set: ");
        for (int i = 0; i < n; i++) {
            scanf("%d", &set[i]);
        }

        printf("Enter the target sum: ");
        scanf("%d", &targetSum);

        printf("Subsets with sum %d:\n", targetSum);
        findSubset(0, 0, 0);

        if (!flag) {
            printf("There is no solution\n");
        }

        return 0;
}
```

Sample Output:

```
Enter the number of elements in the set: 8
Enter the elements of the set: 1 2 3 4 5 6 7 8
Enter the target sum: 8

Subsets with sum 8:

{1, 2, 5}

{1, 3, 4}

{1, 7}

{2, 6}

{3, 5}

{8}
```

OUTPUT:

EXPERIMENT-9

Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

About the Program:

This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.
2. Search the minimum element in the list.
3. Swap with value at location MIN.
4. Increment MIN to point to next element.
5. Repeat until the list is sorted.

Algorithm: Selection-Sort (A)

```
fori← 1 to n-1 do
  min j ←i;
  min x ← A[i]
  for j ←i + 1 to n do
    if A[j] < min x then
      min j ← j
      min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```

Program:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>
```

```
// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap arr[i] and arr[minIndex]
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    printf("[ ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("]\n");
}

int main() {
    int n;
    int arr[n];
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Generate random array
    printf("Generated elements: ");
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Generating random numbers between 0 and 9999
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Measure time taken to sort
    clock_t start = clock();
    selectionSort(arr, n);
    clock_t end = clock();
    double timeTaken = ((double)(end - start) * 1000000) / CLOCKS_PER_SEC; // Convert to
nanoseconds

    // Print sorted array
    printf("Sorted array: ");
    printArray(arr, n);

    // Print time taken
    printf("Time taken to sort %d elements: %.2f nanoseconds\n", n, timeTaken);
    return 0;
}
```

Sample Output:

```
Enter the number of elements: 100
```

```
Generated elements: 4057 1343 9482 2362 2187 6758 3456 8225 1589 118 6668 2092 3858
8877 5519 1357 336 3248 5821 4926 3426 5577 9019 3420 189 4285 3985 3653 8263 9796
5613 8673 7491 1447 7387 6031 8205 7196 4256 6146 7314 924 8238 1172 6154 3757 8882
2842 7006 1055 7768 6784 6632 3139 204 6821 7425 541 6826 2040 6690 8791 7065 4181
238 4453 6564 4796 8001 821 7294 5315 8097 5533 2840 4251 5642 8074 7094 9000 9129
1214 2136 2113 4354 8693 5286 1779 9234 2113 171 5924 7256 7237 6458 7495 8042 3022
8643 6043
Sorted array: [ 118 171 189 204 238 336 541 821 924 1055 1172 1214 1343 1357 1447 1589
1779 2040 2092 2113 2113 2136 2187 2362 2840 2842 3022 3139 3248 3420 3426 3456 3653
3757 3858 3985 4057 4181 4251 4256 4285 4354 4453 4796 4926 5286 5315 5519 5533 5577
5613 5642 5821 5924 6031 6043 6146 6154 6458 6564 6632 6668 6690 6758 6784 6821 6826
7006 7065 7094 7196 7237 7256 7294 7314 7387 7425 7491 7495 7768 8001 8042 8074 8097
8205 8225 8238 8263 8643 8673 8693 8791 8877 8882 9000 9019 9129 9234 9482 9796 ]
Time taken to sort 100 elements: 18.00 nanoseconds
```

<u>OUTPUT</u> (Also Plot a graph of the time taken versus some n>100).

EXPERIMENT -10

Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

About the Program:

The basic idea behind QuickSort is to select a “pivot” element from the array and partition the other elements into two sub-arrays according to whether they are less than or greater than the pivot.

Algorithm:

```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //         indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ←Partition(A[l..r])  //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

```
ALGORITHM Partition(A[l..r])
        p←A[l]
        i ←l; j ←r + 1
        repeat
        repeat i ←i + 1 until A[i]≥ p
        repeat j ←j − 1 until A[j ]≤ p
        swap(A[i], A[j ])
        until i ≥ j
        swap(A[i], A[j ])      //undo last swap when i ≥ j
        swap(A[l], A[j ])
        return j
```

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[low];
    int i = low, j = high;

    while (i < j) {
        while (arr[i] <= pivot && i <= high)
            i++;
        while (arr[j] > pivot && j >= low)
            j--;
        if (i < j) {
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[low] (pivot) and arr[j]
    int temp = arr[low];
    arr[low] = arr[j];
    arr[j] = temp;

    return j;
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        // Sort the elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
    int main() {
        int n;
        int arr[n];
        printf("Enter the number of elements: ");
        scanf("%d", &n);
        // Generate random array
        printf("Generated elements: ");
        srand(time(NULL));
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 10000; // Generating random numbers between 0 and 9999
            printf("%d ", arr[i]);
        }
        printf("\n");

        // Measure time taken to sort
        clock_t start = clock();
        quickSort(arr, 0, n - 1);
        clock_t end = clock();
        double timeTaken = ((double)(end - start) * 1000000) / CLOCKS_PER_SEC; //
       Convert to nanoseconds

        // Print sorted array
        printf("Sorted array: ");
        printf("[ ");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("]\n");

        // Print time taken
        printf("Time taken to sort %d elements: %.2f nanoseconds\n", n, timeTaken);
        return 0;
}
```

Sample Output:

```
    Enter the number of elements: 50
    Generated elements: 6250 5051 734 3676 9468 8346 7568 5516 561 1074 5871 3074
       1498 6984 1977 8072 9911 2648 8569 1442 7671 4351 764 1765 3383 1836 27 6993
       8608 302 8177 1210 5353 5263 4886 1173 3609 2454 6690 522 9880 2561 3597 1379
       5897 5574 5803 5808 4574 4372
    Sorted array: [ 27 302 522 561 734 764 1074 1173 1210 1379 1442 1498 1765 1836 1977
       2454 2561 2648 3074 3383 3597 3609 3676 4351 4372 4574 4886 5051 5263 5353
       5516 5574 5803 5808 5871 5897 6250 6690 6984 6993 7568 7671 8072 8177 8346
       8569 8608 9468 9880 9911 ]
Time taken to sort 50 elements: 5.00 nanoseconds
```

OUTPUT(with graph):

EXPERIMENT-11

Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

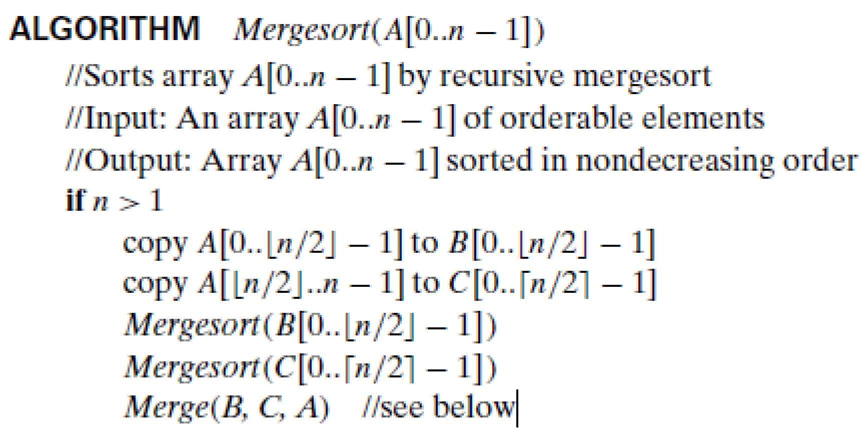
<u>**About the program:**</u>

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

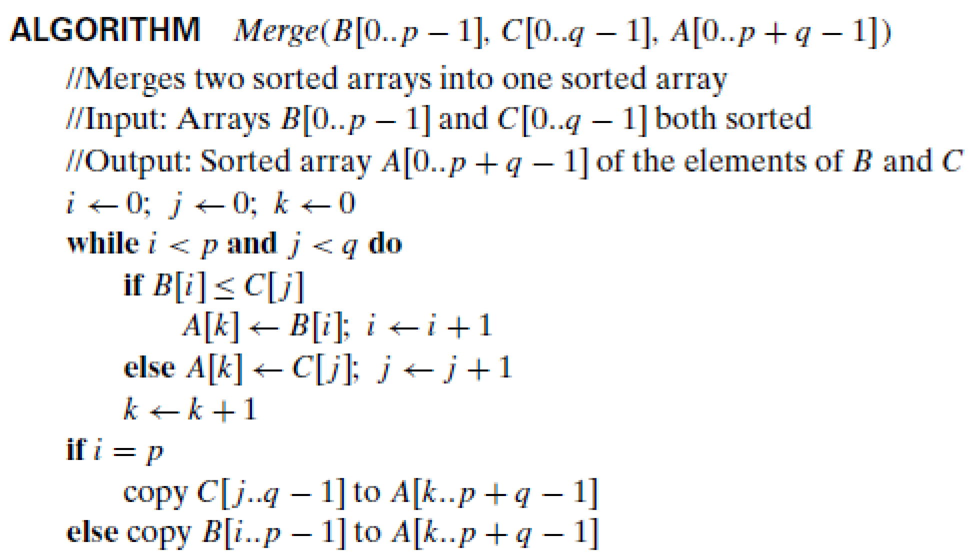
Here’s a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Algorithm:



```
ALGORITHM  Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)    //see below
```



```
ALGORITHM  Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    i ← 0; j ← 0; k ← 0
    while i < p and j < q do
        if B[i] ≤ C[j]
            A[k] ← B[i]; i ← i + 1
        else A[k] ← C[j]; j ← j + 1
        k ← k + 1
    if i = p
        copy C[j..q − 1] to A[k..p + q − 1]
    else copy B[i..p − 1] to A[k..p + q − 1]
```

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Merge function to merge two sorted subarrays
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    // Create temporary arrays
    int L[n1], R[n2];
    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Merge Sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

```
// Function to print an array
void printArray(int arr[], int n) {
    printf("[ ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("]\n");
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Generate random array
    printf("Generated elements: ");
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Generating random numbers between 0 and 9999
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Measure time taken to sort
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();
    double timeTaken = ((double)(end - start) * 1000000) / CLOCKS_PER_SEC; // Convert to
nanoseconds

    // Print sorted array
    printf("Sorted array: ");
    printArray(arr, n);

    // Print time taken
    printf("Time taken to sort %d elements: %.2f nanoseconds\n", n, timeTaken);

    return 0;
}
```

Sample Output:

```
Enter the number of elements: 200
Generated elements: 177 1589 8112 6997 5759 6217 7996 5204 2676 3838 2103 4115 3679
1299 6491 5793 2639 5955 3144 2521 526 7090 1341 1544 6178 7467 9178 2297 9380 4879
608 5910 6468 5072 2907 2227 1290 7256 3783 318 1094 2238 4433 1125 9889 924 6918
2528 3231 62 1401 109 3504 2742 1654 6034 210 832 8331 5942 2063 5291 1852 4883 364
4760 3462 8006 8368 7245 8324 9462 5835 9110 588 5724 34 3858 4604 9618 273 6005 6079
3777 8747 7733 9812 5309 4917 4495 1252 3332 6139 3104 8215 2855 4216 8029 7213 2584
5274 1889 8399 7461 999 5339 3185 7386 9197 7789 7004 9470 3794 3083 3248 8894 7169
9412 4203 8438 259 5455 8123 6398 4912 6338 5605 5480 4368 2818 4417 5994 4708 2816
9808 5707 8155 2993 9445 3704 7135 2801 3175 7281 2237 2775 2527 5758 8539 6731 4196
8798 8538 8671 1549 9802 5010 7154 5283 5730 9973 9700 8076 1033 8868 7884 6740 3375
7230 2538 7079 717 5339 6606 7998 7576 9381 526 3334 7920 3609 3883 6719 2147 2554
4620 8302 3916 1774 3585 5998 1747 9637 427 2780 4857 4663 5873 8232 1893
Sorted array: [ 34 62 109 177 210 259 273 318 364 427 526 526 588 608 717 832 924 999
1033 1094 1125 1252 1290 1299 1341 1401 1544 1549 1589 1654 1747 1774 1852 1889 1893
2063 2103 2147 2227 2237 2238 2297 2521 2527 2528 2538 2554 2584 2639 2676 2742 2775
2780 2801 2816 2818 2855 2907 2993 3083 3104 3144 3175 3185 3231 3248 3332 3334 3375
3462 3504 3585 3609 3679 3704 3777 3783 3794 3838 3858 3883 3916 4115 4196 4203 4216
4368 4417 4433 4495 4604 4620 4663 4708 4760 4857 4879 4883 4912 4917 5010 5072 5204
5274 5283 5291 5309 5339 5339 5455 5480 5605 5707 5724 5730 5758 5759 5793 5835 5873
5910 5942 5955 5994 5998 6005 6034 6079 6139 6178 6217 6338 6398 6468 6491 6606 6719
6731 6740 6918 6997 7004 7079 7090 7135 7154 7169 7213 7230 7245 7256 7281 7386 7461
7467 7576 7733 7789 7884 7920 7996 7998 8006 8029 8076 8112 8123 8155 8215 8232 8302
8324 8331 8368 8399 8438 8538 8539 8671 8747 8798 8868 8894 9110 9178 9197 9380 9381
9412 9445 9462 9470 9618 9637 9700 9802 9808 9812 9889 9973 ]
Time taken to sort 200 elements: 30.00 nanoseconds
```

OUTPUT(With graph):

EXPERIMENT-12

Design and implement C/C++ Program for N Queen's problem using Backtracking.

About the program:

The N-Queens problem is a classic chessboard puzzle where the goal is to place N queens on an N×N chessboard so that no two queens threaten each other. It's a well-known example of a combinatorial optimization problem with applications in computer science, mathematics, and artificial intelligence. Backtracking is a common algorithmic technique used to solve this problem by systematically exploring all possible configurations of queen placements, backtracking whenever a conflict arises until a valid solution is found or all possibilities are exhausted.

Program:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
int totalSolutions = 0; // Global variable to store the total number of solutions

// Function to print the board
void printSolution(int board[], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i] == j)
                printf(" Q ");
            else
                printf(" - ");
        }
        printf("\n");
    }
}

// Function to check if it's safe to place a queen at board[row][col]
bool isSafe(int board[], int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row)) {
            return false;
        }
    }
    return true;
}

// Recursive function to solve N Queen's problem
void solveNQUtil(int board[], int row, int N) {
    if (row >= N) {
        printSolution(board, N);
        printf("\n");
        totalSolutions++;
        return;
    }

    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col)) {
            board[row] = col;
            solveNQUtil(board, row + 1, N);
            board[row] = -1; // backtrack
        }
    }
}

// Main function to solve N Queen's problem
void solveNQ(int N) {
    int board[N];
    for (int i = 0; i < N; i++) {
        board[i] = -1;
    }

    solveNQUtil(board, 0, N);
```

```
}

// Driver program
int main() {
    int N;
    printf("Enter the number of queens: ");
    scanf("%d", &N);

    solveNQ(N);
    printf("Total solutions: %d\n", totalSolutions);
    return 0;
}
```

Sample Output:

```
Enter the number of queens: 4
 -    Q    -    -
 -    -    -    Q
 Q    -    -    -
 -    -    Q    -


 -    -    Q    -
 Q    -    -    -
 -    -    -    Q
 -    Q    -    -


Total solutions: 2
```

OUTPUT:

VIVA-VOCE

1. **What is the significance of algorithm analysis in computer science?**
   Algorithm analysis is crucial in determining the efficiency of algorithms in terms of time and space. It helps in making informed decisions about which algorithm to use for solving a particular problem.

2. **Define time complexity and space complexity.**
   Time complexity refers to the amount of time an algorithm takes to solve a problem as a function of the input size. Space complexity, on the other hand, refers to the amount of memory space an algorithm requires to execute as a function of the input size.

3. **How is time complexity represented using Big O notation?**
   Big O notation provides an upper bound on the time complexity of an algorithm. It represents the worst-case scenario of an algorithm's time complexity as a function of the input size.

4. **Describe the process of analyzing the time complexity of an algorithm.**
   Time complexity analysis involves determining the number of basic operations executed by an algorithm as a function of the input size. This is typically done by counting the number of iterations in loops, recursive calls, or other fundamental operations.

5. **What is the difference between best-case and worst-case time complexity?**
   Best-case time complexity represents the minimum number of operations an algorithm performs for a given input size, while worst-case time complexity represents the maximum number of operations.

6. **How do you calculate the space complexity of an algorithm?**
   Space complexity analysis involves determining the maximum amount of memory space required by an algorithm to execute as a function of the input size. This includes memory used by variables, data structures, and function calls.

7. **Explain the concept of asymptotic analysis.**
   Asymptotic analysis focuses on the behavior of an algorithm as the input size approaches infinity. It ignores constant factors and lower-order terms, focusing on the dominant term to describe the growth rate of the algorithm's time or space complexity.

8. **What are the characteristics of a greedy algorithm?**
   Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. They are often simple, efficient, and easy to implement, but may not always provide an optimal solution.

9. **How do you solve recurrence relations?**
   Recurrence relations are solved using various techniques such as substitution, recursion tree method, and master theorem. These methods help in finding closed-form solutions for the recurrence relation.

10. **Discuss the significance of dynamic programming in algorithm design.**
    Dynamic programming is a powerful technique used to solve problems by breaking them down into simpler sub problems and storing the solutions to avoid redundant computations. It is particularly useful for optimization problems with overlapping sub problems.

11. **What is the principle of optimality in dynamic programming?**
    The principle of optimality states that an optimal solution to a problem contains optimal

solutions to its sub problems. In other words, if we can solve smaller sub problems optimally, we can combine these solutions to obtain the optimal solution to the larger problem.

12. **Explain the divide and conquer algorithm design paradigm.**
    Divide and conquer is a problem-solving technique where a problem is broken down into smaller sub problems that are easier to solve. These sub problems are solved recursively, and then their solutions are combined to solve the original problem.

13. **What is backtracking, and when is it used in algorithm design?**
    Backtracking is a technique used to systematically search for solutions to a problem by exploring all possible candidates. It is used when the problem can be broken down into a sequence of decisions, and it is not feasible to try all possible solutions exhaustively.

14. **Discuss the difference between greedy algorithms and dynamic programming.**
    Greedy algorithms make locally optimal choices at each step without considering the overall structure of the problem, while dynamic programming breaks down a problem into smaller sub problems and stores the solutions to avoid redundant computations. Greedy algorithms do not always guarantee an optimal solution, whereas dynamic programming does when certain conditions are met.

15. **What are the key components of a dynamic programming algorithm?**
    The key components of a dynamic programming algorithm include defining the sub problems, identifying the recurrence relation, memorization or tabulation to store and reuse solutions to sub problems, and constructing the final solution from the computed solutions.

16. **How do you evaluate the efficiency of an algorithm in practical scenarios?**
    The efficiency of an algorithm in practical scenarios can be evaluated by measuring its runtime performance using timing experiments on various input sizes. Additionally, analyzing its space complexity and comparing it with other algorithms for the same problem can also provide insights into its efficiency.

17. **Discuss the trade-offs involved in selecting an algorithm for a given problem.**
    The trade-offs involved in selecting an algorithm include considerations such as time complexity, space complexity, ease of implementation, and optimality of the solution. Some algorithms may have better time complexity but higher space complexity, while others may sacrifice optimality for simplicity.

18. **How do you improve the efficiency of algorithms?**
    Efficiency of algorithms can be improved by optimizing their time complexity through algorithmic improvements such as using more efficient data structures, reducing redundant computations, or applying problem-specific optimizations. Additionally, parallelization, caching, and hardware-specific optimizations can also enhance performance.

19. **Discuss the concept of memorization in dynamic programming.**
    Memorization is a technique used to store the solutions to sub problems in dynamic programming to avoid redundant computations. It involves storing the results of solved sub problems in a data structure (such as an array or hash table) so that they can be reused when needed.

20. **What is the significance of recurrence relations in algorithm analysis?**
    Recurrence relations are used to describe the time complexity of recursive algorithms by expressing the time taken to solve a problem of size n in terms of the time taken to solve smaller sub problems. Solving recurrence relations helps in analyzing the time complexity of recursive

algorithms and designing efficient algorithms.

21. **What is a spanning tree?**
    A spanning tree of a connected graph is a subgraph that is a tree, containing all the vertices of the original graph and a subset of its edges.

22. **What is Kruskal's algorithm used for?**
    Kruskal's algorithm is used to find the Minimum Spanning Tree (MST) of a given connected, undirected graph.

23. **How does Kruskal's algorithm work?**
    Kruskal's algorithm works by greedily selecting edges with the smallest weights while ensuring that no cycles are formed until all vertices are included in the MST.

24. **What is the time complexity of Kruskal's algorithm?**
    The time complexity of Kruskal's algorithm is O(E log V), where E is the number of edges and V is the number of vertices in the graph.

25. **What is Prim's algorithm used for?**
    Prim's algorithm is used to find the Minimum Spanning Tree (MST) of a given connected, undirected graph.

26. **How does Prim's algorithm work?**
    Prim's algorithm works by starting with an arbitrary vertex and greedily adding the shortest edge that connects a vertex in the MST to a vertex outside the MST until all vertices are included in the MST.

27. **What is the time complexity of Prim's algorithm?**
    The time complexity of Prim's algorithm is O(V^2) with an adjacency matrix representation or O(E + V log V) with an adjacency list representation.

28. **What is the All-Pairs Shortest Paths problem?**
    The All-Pairs Shortest Paths problem involves finding the shortest path between every pair of vertices in a weighted graph.

29. **How does Floyd's algorithm work?**
    Floyd's algorithm iteratively considers all vertices as intermediate vertices and calculates the shortest path between every pair of vertices.

30. **What is the time complexity of Floyd's algorithm?**
    The time complexity of Floyd's algorithm is O(V^3), where V is the number of vertices in the graph.

31. **What is the transitive closure of a graph?**
    The transitive closure of a graph determines whether there is a path between every pair of vertices in the graph.

32. **How does Warshall's algorithm work?**
    Warshall's algorithm computes the transitive closure of a graph using dynamic programming.

33. **What is the time complexity of Warshall's algorithm?**
    The time complexity of Warshall's algorithm is O(V^3), where V is the number of vertices in the graph.

34. **What is Dijkstra's algorithm used for?**

Dijkstra's algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph with non-negative edge weights.

35. **How does Dijkstra's algorithm work?**
Dijkstra's algorithm works by greedily selecting the vertex with the smallest distance from the source and updating the distances to its neighboring vertices.

36. **What is the time complexity of Dijkstra's algorithm?**
The time complexity of Dijkstra's algorithm is O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph.

37. **What is a topological ordering of vertices in a directed graph?**
A topological ordering is a linear ordering of vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

38. **When can a topological ordering not be defined for a directed graph?**
A topological ordering cannot be defined for a directed graph containing a cycle.

39. **How is a topological ordering obtained in a directed graph?**
A topological ordering can be obtained using algorithms such as Depth-First Search (DFS).

40. **What is the 0/1 Knapsack Problem?**
The 0/1 Knapsack Problem is a classic optimization problem where the goal is to maximize the value of items selected, subject to a constraint on the total weight that can be carried.

41. **How does the dynamic programming approach solve the 0/1 Knapsack Problem?**
The dynamic programming approach builds a table to store the maximum value that can be achieved with different capacities of the knapsack, considering each item individually.

42. **What is the time complexity of the dynamic programming solution for the 0/1 Knapsack Problem**?
The time complexity of the dynamic programming solution for the 0/1 Knapsack Problem is O(nW), where n is the number of items and W is the capacity of the knapsack.

43. **How does the greedy approximation method solve the Knapsack Problem?**
The greedy approximation method selects items based on a greedy criterion (such as value-to-weight ratio) without considering the global optimum, which may lead to a suboptimal solution.

44. **What are the differences between the discrete and continuous Knapsack Problems?**
In the discrete Knapsack Problem, each item can be selected either completely or not at all, while in the continuous Knapsack Problem, fractions of items can be selected.

45. **Does the greedy approximation method guarantee an optimal solution for the Knapsack Problem?**
No, the greedy approximation method does not guarantee an optimal solution for the Knapsack Problem, especially for the discrete Knapsack Problem.

46. **What is the Subset Sum Problem?**
The Subset Sum Problem is a decision problem that asks whether there exists a subset of a given set whose elements sum up to a specified target value.

47. **How can the Subset Sum Problem be solved using dynamic programming?**
The Subset Sum Problem can be solved using dynamic programming by building a table to store whether it's possible to achieve each sum using a subset of the given elements.

48. **What is the time complexity of the dynamic programming solution for the Subset Sum**

Problem?
The time complexity of the dynamic programming solution for the Subset Sum Problem is O(nS), where n is the number of elements in the set and S is the target sum.

49. **What are the sorting algorithms you've implemented?**
The sorting algorithms implemented are Selection Sort, Quick Sort, and Merge Sort.

50. **How does each sorting algorithm work?**
Selection Sort repeatedly selects the smallest element and swaps it with the element in the current position. Quick Sort partitions the array around a pivot element and recursively sorts the partitions. Merge Sort divides the array into two halves, recursively sorts them, and then merges the sorted halves.

51. **What are the time complexities of these sorting algorithms?**
The time complexity of Selection Sort is O(n^2), Quick Sort is O(n log n) on average but O(n^2) in the worst case, and Merge Sort is O(n log n).

52. **What is the N Queen's Problem?**
The N Queen's Problem is a classic problem of placing N chess queens on an N×N chessboard in such a way that no two queens attack each other.

53. **How does backtracking solve the N Queen's Problem?**
Backtracking is a systematic way to explore all possible configurations of queen placements on the chessboard, discarding those configurations that violate the constraint of no two queens attacking each other.

54. **What is the time complexity of the backtracking solution for the N Queen's Problem?**
The time complexity of the backtracking solution for the N Queen's Problem is exponential, but it can be optimized using pruning techniques.

****