



REVA
UNIVERSITY
Bengaluru, India



**SCHOOL
OF
COMPUTER SCIENCE
AND
ENGINEERING**

**ALGORITHMS LAB
B22EF0404**

Fourth Semester
AY-2023-24
(Prepared in Feb-2024)

INDEX

SL. No	Contents	Page. no
1	Lab Objectives	3
2	Lab Outcomes	3
3	Lab Requirements	3
4	Guidelines to Students	4
5	List of Lab Exercises and Mini-Projects	5
6	Lab Exercises' Solutions:	7
PART – A: LAB EXERCISES		
	Session – 1: Lab Exercise	7
	Session – 2a: Lab Exercise	10
	Session – 2b: Lab Exercise	15
	Session – 3: Lab Exercise	17
	Session – 4: Lab Exercise	19
	Session – 5: Lab Exercise	22
	Session – 6: Lab Exercise	24
	Session – 7: Lab Exercise	27
	Session – 8a: Lab Exercise	30
	Session – 8b: Lab Exercise	32
	Session – 9: Lab Exercise	33
	Session – 10: Lab Exercise	36
PART – B: MINI-PROJECTS		
7	Mini-Project Specification and Evaluation Rubrics	40
8	Project Report Format	43
9	Learning Resources	48

1. Lab Objectives:

The objectives of this course are to

- Demonstrate performance of algorithms with respect to time and space complexity.
- Explain Graph and Tree traversals techniques.
- Understand the concepts of greedy method and dynamic programming for different applications.
- Illustrate the methods of Backtracking, Branch and bound techniques.
- Familiarize the concepts of deterministic and non-deterministic algorithms.

2. Lab Outcomes:

On successful completion of this course; student shall be able to:

CO#	Course Outcomes	POs	PSOs
CO1	Implement sorting and searching techniques using different algorithmic techniques.	1, 2, 3, 5, 9,12	1,3
CO2	Implement Tree Traversal method and Greedy Algorithms	1, 2, 3, 5, 9,12	1,3
CO3	Develop the skill set to solve problems using Dynamic Programming concepts.	1, 2, 3, 5, 6, 9, 12	2,3
CO4	Illustrate Backtracking, Branch and Bound concept to solve various problems	1, 2, 3, 5, 6, 9, 12	1,2
CO5	Demonstrate Time and Space complexities of various algorithms	1, 2, 3, 5, 6, 9, 12	1,2
CO6	Analyze and evaluate different performance analysis methods for non-deterministic algorithms	1, 2, 3,4, 5, 6, 9, 12	2,3

3. Lab Requirements:

The following are the required hardware and software for this lab.

Hardware Requirements: A standard personal computer or laptop with the following minimum specifications:

1. **Processor:** Any modern multi-core processor (e.g., Intel Core i5 or AMD Ryzen series).
2. **RAM:** At least 4 GB of RAM for basic programs; 8 GB or more is recommended for larger data structures and complex algorithms.
3. **Storage:** A few gigabytes of free disk space for the development environment and program files.
4. **Display:** A monitor with a resolution of 1024x768 or higher is recommended for comfortable coding.
5. **Input Devices:** Keyboard and mouse (or other input devices) for coding and interacting with the development environment.

Software Requirements:

1. **Operating System:** You can develop and execute C programs for Algorithms Lab on various operating systems, including Windows, macOS, and Linux.
2. **Text Editor or Integrated Development Environment (IDE):**

- **Text Editor:** You can use a simple text editor like Notepad (Windows), Nano (Linux/macOS), or any code-oriented text editor like Visual Studio Code, Sublime Text, or Atom.
 - **IDE:** Consider using a C/C++ integrated development environment like Code::Blocks, Dev-C++, or Eclipse CDT for a more feature-rich coding experience.
3. **C Compiler:**
- To compile and execute C programs, you need a C compiler. GCC (GNU Compiler Collection) is a popular choice for Linux and macOS.
 - For Windows, you can use MinGW (Minimalist GNU for Windows) with GCC or Visual C++ from Microsoft if you prefer a Microsoft development environment.

4. Guidelines to Students:

- Equipment in the lab for the use of the student community. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage caused is punishable.
- Students are required to carry their observation / programs book with completed exercises while entering the lab.
- Students are supposed to occupy the machines allotted to them and are not supposed to talk or make noise in the lab. The allocation is put up on the lab notice board.
- The lab can be used in free time / lunch hours by the students who need to use the systems should get prior permission from the lab in-charge.
- Lab records need to be submitted on or before the date of submission.
- Students are not supposed to use flash drives.

Practice

Here are some key activities involved in the design and analysis of algorithms:

- **Problem Understanding:** The first step is to clearly understand the problem at hand. This involves identifying the input, output, constraints, and any specific requirements.
- **Algorithm Design:** Once the problem is understood, the next step is to design an algorithm to solve it. This involves creating a step-by-step procedure or a set of rules that outlines how the problem can be solved.
- **Algorithm Analysis:** After designing the algorithm, it is essential to analyze its efficiency and performance. This analysis includes evaluating the algorithm's time complexity (how the running time grows as the input size increases) and space complexity (how much memory the algorithm requires).
- **Algorithm Optimization:** Based on the analysis, the algorithm can be optimized to improve its efficiency. This may involve making algorithmic modifications, utilizing data structures that are better suited for the problem, or applying known optimization techniques.
- **Algorithm Correctness:** Ensuring the algorithm is correct is crucial. This involves proving its correctness using formal methods like mathematical proofs or conducting extensive testing to verify that the algorithm produces the correct output for different input scenarios.
- **Algorithm Implementation:** Once the algorithm design is finalized and its correctness is established, the next step is to implement the algorithm in a programming language. This involves translating the algorithm into code and addressing any specific programming language considerations.
- **Experimental Analysis:** To validate the algorithm's performance in practice, experimental analysis can be conducted. This involves running the algorithm on various inputs and measuring its running time, memory usage, and other relevant metrics. The results can be compared with the theoretical analysis to verify the algorithm's efficiency.

These activities are iterative and may involve revisiting earlier stages as the design and analysis process progresses. The goal is to create efficient and reliable algorithms that can effectively solve specific problems.

5. List of Lab Exercises and Mini-Projects:

Sl No.	Title of the Exercise
Part A	
1	Sort a given set of elements using the Quicksort method 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. The elements can be read from a file or can be generated using the random number generator.
2	a. Obtain the Topological ordering of vertices in a given digraph. b. Compute the transitive closure of a given directed graph using Warshall's algorithm
3	Implement 0/1 Knapsack problem using Dynamic Programming.
4	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.
5	Find the Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.
6	Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.
7	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.
8	For a given graph a. Print all the nodes reachable from a given starting node in a digraph using BFS method. b. Check whether a given graph is connected or not using DFS method.
9	Implement N Queen's problem using Back Tracking
10	Implement All-Pairs Shortest Paths Problem using Floyd's algorithm
Part B	
1	Mini Projects on Sorting Algorithm Efficiency Comparison.
2	Mini Projects on Dynamic Programming
3	Mini Projects on Mini Projects on Pathfinding
4	Mini Projects on Graph Traversal Techniques
5	Mini Projects on Traveling Salesman Problem

PART – A

LAB EXERCISES

6. Lab Exercises' Solutions:

	Solutions (Part A)
1.	<p>Sort a given set of elements using the Quicksort method 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. The elements can be read from a file or can be generated using the random number generator.</p> <p>Problem Statement: The task is to sort a given set of elements using the Quicksort method and measure the time it takes to perform the sorting. The experiment should be repeated for different values of n, representing the number of elements in the list to be sorted. Finally, a graph should be plotted, showing the relationship between the time taken and the number of elements (n). The elements can either be read from a file or generated using a random number generator.</p> <p>Solution Overview: The solution involves implementing the Quicksort algorithm to sort the given set of elements. The time taken for sorting will be measured using appropriate time-tracking functions. The experiment will be repeated for various values of n, allowing the observation of how the sorting time scales with the number of elements.</p> <p>Intuition: Quicksort is a divide-and-conquer algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The process is then applied recursively to the sub-arrays. The efficiency of Quicksort makes it a popular choice for sorting large datasets.</p> <p>Code Implementation: This following program generates random arrays of different sizes, sorts them using the Quicksort method, and prints the time taken for each experiment. You can use the collected data to plot a graph of time taken versus the number of elements (n).</p> <pre>#include <stdio.h> #include <stdlib.h> #include <time.h> // Function to partition the array int partition(int arr[], int low, int high) { int pivot = arr[high]; int i = low - 1; for (int j = low; j < high; j++) { if (arr[j] <= pivot) { i++; // Swap arr[i] and arr[j] int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp; } } // Swap arr[i+1] and arr[high] (pivot) int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp; return i + 1; } // Quicksort algorithm</pre>

```
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        // Find pivot such that elements smaller than pivot are on the left,
        // and elements greater than pivot are on the right
        int pivotIndex = partition(arr, low, high);

        // Recursively sort the sub-arrays
        quicksort(arr, low, pivotIndex - 1);
        quicksort(arr, pivotIndex + 1, high);
    }
}

int main() {
    srand(time(0)); // Seed for random number generation
    clock_t start, end;

    // Experiment for different values of n
    for (int n = 1000; n <= 10000; n += 1000) {
        int arr[n];

        // Generate random elements
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 1000; // Random numbers between 0 and 999
        }

        // Measure the time taken for sorting
        start = clock();
        quicksort(arr, 0, n - 1);
        end = clock();

        double timeTaken = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Print the results
        printf("For n = %d, Time Taken: %lf seconds\n", n, timeTaken);
    }

    return 0;
}
```

Sample Output:

```
For n = 1000, Time Taken: 0.000108 seconds
For n = 2000, Time Taken: 0.000143 seconds
For n = 3000, Time Taken: 0.000221 seconds
For n = 4000, Time Taken: 0.000300 seconds
For n = 5000, Time Taken: 0.000393 seconds
For n = 6000, Time Taken: 0.000452 seconds
For n = 7000, Time Taken: 0.000560 seconds
For n = 8000, Time Taken: 0.000666 seconds
For n = 9000, Time Taken: 0.000712 seconds
For n = 10000, Time Taken: 0.000833 seconds
```

```
=== Code Execution Successful ===
```


Outcome of the Exercise:

The implementation and experimentation with the Quicksort algorithm yielded valuable insights into the efficiency of the sorting process. The key observations and outcomes are summarized below:

1. Algorithm Efficiency:

- Quicksort demonstrated efficient sorting, especially for large datasets. The divide-and-conquer approach of Quicksort allows for effective handling of diverse data sizes.

2. Scalability Analysis:

- The experiment involved sorting different-sized arrays (n) to observe how the algorithm scales with increasing input sizes. The time taken remained reasonably low even as the number of elements increased, showcasing the scalability of Quicksort.

3. Time Complexity Analysis:

- The observed time complexities align with the expected average-case time complexity of Quicksort, which is $O(n \log n)$. This reinforces the algorithm's suitability for real-world applications where efficient sorting is crucial.

4. Random Element Generation:

- The program's ability to generate random elements for testing provides a realistic simulation of sorting scenarios. Random data sets help evaluate the algorithm's performance under various input conditions.

5. Graphical Representation:

- The plotted graph of time taken versus the number of elements (n) provides a visual representation of the algorithm's performance trends. Such visualizations are essential for understanding the algorithm's behavior as input sizes change.

6. Practical Application:

- The exercise emphasizes the practical application of sorting algorithms, addressing scenarios where datasets need to be organized efficiently. Quicksort's effectiveness is evident in its quick response to varying data sizes.

7. Experimental Flexibility:

- The flexibility to experiment with different input sizes and types of elements (random or read from a file) adds versatility to the analysis. This adaptability allows for a comprehensive exploration of the algorithm's behavior.

8. Learning Experience:

- The exercise serves as a hands-on learning experience, allowing you to apply theoretical knowledge in a practical setting. Implementing the algorithm, measuring its performance, and analyzing the outcomes contribute to a deeper understanding of sorting algorithms.

Viva/Interview Questions:**1. How does the Quicksort algorithm work?**

- *Answer:* Quicksort is a divide-and-conquer algorithm that works by selecting a pivot element and partitioning the other elements into two sub-arrays based on whether they are less than or greater than the pivot. The process is applied recursively to the sub-arrays.

2. What is the time complexity of Quicksort?

- *Answer:* The average-case time complexity of Quicksort is $O(n \log n)$, making it an efficient sorting algorithm. However, in the worst-case scenario, it can degrade to $O(n^2)$ if the pivot selection leads to unbalanced partitions.

3. How does the choice of the pivot element affect Quicksort's performance?

- *Answer:* The choice of the pivot element significantly impacts the efficiency of Quicksort. A well-selected pivot leads to balanced partitions, ensuring better performance. Common strategies include selecting the first, last, or middle element, or using a random element.

4. What is the significance of partitioning in the Quicksort algorithm?

- *Answer:* Partitioning is a fundamental step in Quicksort, where elements are rearranged based on their relationship to the pivot. It divides the array into two sub-arrays, allowing the algorithm to sort each sub-array independently.

5. How does Quicksort compare to other sorting algorithms, such as Merge Sort or Bubble Sort?

- *Answer:* Quicksort is often more efficient than Bubble Sort and comparable to Merge Sort in terms of average-case time complexity. Quicksort is an in-place sorting algorithm, making it memory-efficient, but its worst-case time complexity is a consideration compared to Merge Sort.

6. Explain the concept of the 'partition' function in the Quicksort algorithm.

	<ul style="list-style-type: none"> <i>Answer:</i> The partition function is responsible for rearranging elements in the array such that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right. It returns the index of the pivot after the partitioning process. <p>7. Why is Quicksort preferred for large datasets?</p> <ul style="list-style-type: none"> <i>Answer:</i> Quicksort's average-case time complexity of $O(n \log n)$ and in-place sorting nature make it well-suited for large datasets. Its efficiency stems from the divide-and-conquer strategy, enabling it to quickly sort data by recursively dividing and conquering sub-arrays. <p>8. How can you handle duplicate elements in Quicksort?</p> <ul style="list-style-type: none"> <i>Answer:</i> Handling duplicate elements involves modifying the partitioning logic. Instead of considering elements less than or greater than the pivot, you can create partitions for elements equal to the pivot. This ensures the correct placement of duplicate elements during the sorting process. <p>9. What are some strategies for selecting the pivot element in Quicksort?</p> <ul style="list-style-type: none"> <i>Answer:</i> Strategies include selecting the first, last, or middle element as the pivot. Another approach is to choose a random element. Each strategy has its advantages and potential drawbacks, and the choice can impact the efficiency of the algorithm. <p>10. How can you optimize Quicksort for nearly sorted data?</p> <ul style="list-style-type: none"> <i>Answer:</i> For nearly sorted data, choosing the middle element as the pivot might be less efficient. Instead, selecting the median of three elements (first, middle, and last) or using insertion sort for small sub-arrays can optimize Quicksort's performance.
2	<p>a. Obtain the Topological ordering of vertices in a given digraph.</p> <p>Problem Statement:</p> <p>The task is to obtain the topological ordering of vertices in a directed graph (digraph). Topological ordering is a linear ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering. The goal is to represent the dependencies among vertices.</p> <p>Solution Overview:</p> <p>The solution involves using Depth-First Search (DFS) to perform a topological sort of the vertices. The algorithm explores each vertex and recursively visits its adjacent vertices, ensuring that vertices are visited before their dependent vertices.</p> <p>Intuition:</p> <p>Topological sorting is based on the idea that vertices with no incoming edges (in-degree zero) should come first in the ordering. The DFS approach helps explore the graph, visiting vertices and marking them as visited. The ordering is determined based on the finishing times of the vertices during the DFS traversal.</p> <p>Code Implementation (Uses the Adjacency Matrix to represent Digraph):</p> <pre>#include <stdio.h> #define MAX_VERTICES 100 int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES]; int visited[MAX_VERTICES]; int result[MAX_VERTICES]; int numVertices; // Function to perform depth-first search for topological sorting void topologicalSortDFS(int vertex, int* index) { visited[vertex] = 1;</pre>

```
for (int i = 0; i < numVertices; i++) {
    if (adjacencyMatrix[vertex][i] && !visited[i]) {
        topologicalSortDFS(i, index);
    }
}

result[*index] = vertex;
(*index)--;
}

// Function to perform topological sorting
void topologicalSort() {
    int index = numVertices - 1;

    for (int i = 0; i < numVertices; i++) {
        if (!visited[i]) {
            topologicalSortDFS(i, &index);
        }
    }

    // Print the result in order
    printf("Topological Ordering: ");
    for (int i = 0; i < numVertices; i++) {
        printf("%d ", result[i]);
    }
}

int main() {
    // Input number of vertices
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &numVertices);

    // Initialize adjacency matrix
    for (int i = 0; i < numVertices; i++) {
        visited[i] = 0;
        for (int j = 0; j < numVertices; j++) {
            adjacencyMatrix[i][j] = 0;
        }
    }
}
```

```
// Input number of edges
int numEdges;
printf("Enter the number of edges in the graph: ");
scanf("%d", &numEdges);

// Input edges
printf("Enter the edges (vertex1 vertex2):\n");
for (int i = 0; i < numEdges; i++) {
    int start, end;
    scanf("%d %d", &start, &end);
    adjacencyMatrix[start][end] = 1;
}

// Perform topological sorting
topologicalSort();

return 0;
}
```

Sample Output:

```
Enter the number of vertices in the graph: 6
Enter the number of edges in the graph: 6
Enter the edges (vertex1 vertex2):
5 2
5 0
4 0
4 1
2 3
3 1
Topological Ordering: 5 4 2 3 1 0

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully obtains the topological ordering of vertices in the given directed graph. It demonstrates the application of depth-first search to identify the linear ordering that satisfies the dependencies between vertices.

Interview Questions:

1. What is topological sorting, and when is it used?

	<ul style="list-style-type: none"> <i>Answer:</i> Topological sorting is a linear ordering of vertices in a directed graph such that for every directed edge (u, v), vertex u comes before v in the ordering. It is used to represent dependencies between tasks or events. <p>2. Explain the significance of in-degree in topological sorting.</p> <ul style="list-style-type: none"> <i>Answer:</i> In-degree represents the number of incoming edges to a vertex. In topological sorting, vertices with in-degree zero are considered starting points for the ordering since they have no dependencies. <p>3. How does depth-first search contribute to topological sorting?</p> <ul style="list-style-type: none"> <i>Answer:</i> Depth-first search explores the graph, marking vertices as visited. During the traversal, finishing times are recorded, and the vertices are arranged in reverse order of finishing times to achieve a topological ordering. <p>4. What is the time complexity of the topological sorting algorithm?</p> <ul style="list-style-type: none"> <i>Answer:</i> The time complexity depends on the traversal algorithm. For depth-first search-based topological sorting, the time complexity is $O(V + E)$, where V is the number of vertices, and E is the number of edges in the graph. <p>5. Can topological sorting be applied to a graph with cycles?</p> <ul style="list-style-type: none"> <i>Answer:</i> No, topological sorting is not possible in a graph with cycles because the dependencies become cyclic, making it impossible to define a linear ordering. <p>6. How does the program handle multiple valid topological orderings?</p> <ul style="list-style-type: none"> <i>Answer:</i> The program prints one valid topological ordering. However, there can be multiple valid orderings. The choice depends on the order in which vertices with in-degree zero are processed during the DFS traversal. <p>7. Explain the role of in-degree in detecting the starting vertices for topological sorting.</p> <ul style="list-style-type: none"> <i>Answer:</i> In-degree helps identify vertices with no dependencies (in-degree zero), making them suitable starting points for topological sorting. These vertices can be processed first, followed by their dependent vertices. <p>8. Can topological sorting be applied to an undirected graph?</p> <ul style="list-style-type: none"> <i>Answer:</i> Topological sorting is specific to directed acyclic graphs (DAGs). It cannot be directly applied to undirected graphs or graphs with cycles. <p>9. How can topological sorting be used in real-world applications?</p> <ul style="list-style-type: none"> <i>Answer:</i> Topological sorting finds applications in task scheduling, course prerequisites, and job scheduling, where tasks or events have dependencies that need to be satisfied in a specific order. <p>10. How does the program handle the scenario of a disconnected graph?</p> <ul style="list-style-type: none"> <i>Answer:</i> The program identifies connected components in the graph and applies topological sorting separately to each component. The in-degree is considered only for the vertices in the current connected component.
2	<p>b. Compute the transitive closure of a given directed graph using Warshall's algorithm</p> <p>Problem Statement: The problem is to compute the transitive closure of a given directed graph. Transitive closure is a matrix representation that indicates whether there is a path between every pair of vertices in the graph.</p> <p>Solution Overview: The solution involves applying Warshall's algorithm to determine the transitive closure of the directed graph. Warshall's algorithm is based on the concept of matrix multiplication and iterative closure computation.</p> <p>Intuition: Warshall's algorithm works by iteratively updating a matrix to capture the transitive closure. The key idea is to</p>

check for the existence of a path between every pair of vertices through an intermediate vertex. The algorithm repeats this process until the closure is complete.

Code Implementation:

```
#include <stdio.h>

#define MAX_VERTICES 100

// Function to find the transitive closure using Warshall's Algorithm
void transitiveClosure(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    int i, j, k;

    for (k = 0; k < vertices; k++) {
        for (i = 0; i < vertices; i++) {
            for (j = 0; j < vertices; j++) {
                // Update the transitive closure matrix
                graph[i][j] = graph[i][j] || (graph[i][k] && graph[k][j]);
            }
        }
    }
}

// Function to display the adjacency matrix
void displayMatrix(int matrix[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Transitive Closure Matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int vertices, i, j;

    // Read the number of vertices in the graph
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &vertices);

    // Declare an adjacency matrix
    int graph[MAX_VERTICES][MAX_VERTICES];

    // Read the adjacency matrix
    printf("Enter the adjacency matrix (1 for an edge, 0 otherwise):\n");
    for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    // Calculate transitive closure using Warshall's Algorithm
```

```
transitiveClosure(graph, vertices);

// Display the transitive closure matrix
displayMatrix(graph, vertices);

return 0;
}
```

Sample output:

```
Enter the number of vertices in the graph: 4
Enter the adjacency matrix (1 for an edge, 0 otherwise):
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
Transitive Closure Matrix:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully computes the transitive closure of the given directed graph using Warshall's algorithm. The transitive closure matrix indicates whether there is a path between every pair of vertices.

Interview Questions:

- 1. What is the significance of the transitive closure in a directed graph?**
 - Answer:* The transitive closure matrix indicates whether there is a path between every pair of vertices in a directed graph, providing information about the reachability of vertices.
- 2. Explain the concept of Warshall's algorithm and its application in transitive closure computation.**
 - Answer:* Warshall's algorithm iteratively updates a matrix to capture the transitive closure of a directed graph. It checks for the existence of a path between every pair of vertices through an intermediate vertex.
- 3. How does the program handle the initialization of the transitive closure matrix?**
 - Answer:* The program initializes the transitive closure matrix with the original adjacency matrix, representing direct edges between vertices.
- 4. What is the time complexity of Warshall's algorithm for computing the transitive closure?**
 - Answer:* The time complexity is $O(V^3)$, where V is the number of vertices in the graph, due to the three nested loops used for matrix multiplication in the algorithm.
- 5. Can Warshall's algorithm be applied to a graph with cycles?**
 - Answer:* Yes, Warshall's algorithm can be applied to a graph with cycles. It computes the transitive closure even in the presence of cycles.
- 6. How can the transitive closure matrix be interpreted in terms of graph reachability?**
 - Answer:* If the entry `transitiveClosure[i][j]` is 1, it indicates that there is a path from vertex i to vertex j in the graph.
- 7. Can the transitive closure matrix be used to detect strongly connected components in a directed graph?**

	<ul style="list-style-type: none"> • <i>Answer:</i> Yes, the transitive closure matrix can be used to identify strongly connected components. Diagonal elements with a value of 1 represent strongly connected components. <p>8. How does the transitive closure matrix change if a new edge is added to the original graph?</p> <ul style="list-style-type: none"> • <i>Answer:</i> If a new edge is added, the transitive closure matrix is updated to reflect the new reachability information, ensuring that the closure remains accurate. <p>9. What modifications would be needed if the graph is represented using an adjacency list instead of an adjacency matrix?</p> <ul style="list-style-type: none"> • <i>Answer:</i> The algorithm would need adjustments to traverse the adjacency list and update the transitive closure matrix accordingly. <p>10. In which scenarios is computing the transitive closure of a graph useful?</p> <ul style="list-style-type: none"> • <i>Answer:</i> Computing the transitive closure is useful in scenarios where determining reachability between pairs of vertices is essential, such as in network routing, program analysis, and optimization problems.
3	<p>Implement 0/1 Knapsack problem using Dynamic Programming.</p> <p>Problem Statement: The 0/1 Knapsack problem involves selecting a combination of items, each with a specific weight and value, to maximize the total value within a given weight capacity. The constraint is that each item can either be included (1) or excluded (0) from the knapsack.</p> <p>Solution Overview: The dynamic programming approach is used to solve the 0/1 Knapsack problem. The solution involves creating a table to store the maximum value that can be obtained for different subproblems. The final entry in the table represents the maximum value achievable with the given weight capacity.</p> <p>Intuition: The key idea is to build up the solution incrementally by considering each item and either including it in the knapsack or excluding it. The dynamic programming table is filled based on optimal subproblem solutions, considering the maximum value that can be obtained with the current weight capacity and the available items.</p> <p>Code Implementation:</p> <pre>#include <stdio.h> // A utility function that returns maximum of two integers int max(int a, int b) { return (a > b) ? a : b; } // Returns the maximum value that // can be put in a knapsack of capacity W void knapSack(int W, int wt[], int val[], int n) { int i, j; int K[n + 1][W + 1]; // Build table K[][] in bottom up manner for (i = 0; i <= n; i++) { for (j = 0; j <= W; j++) { if (i == 0 j == 0) K[i][j] = 0; else if (wt[i - 1] <= j) K[i][j] = max(K[i - 1][j], val[i - 1] + K[i - 1][j - wt[i - 1]]); else</pre>


```
        K[i][j] = K[i - 1][j];
    }
}
// Print the solution table
printf("\n The solution table:\n");
for(i=0;i<=n;i++)
{
    for(j=0;j<=W;j++)
        printf("%d\t", K[i][j]);
    printf("\n");
}

i = n;
j = W;

printf("\n The Selected Items: ");
while (i > 0 && j > 0) {
    if (K[i][j] != K[i - 1][j]) {
        printf("Item-%d\t", i);
        j -= wt[i - 1];
    }
    i--;
}
printf("\n");
printf("\n The Maximum profit gained = %d",K[n][W]);
}

// Main Program
int main()
{
    int val[10], wt[10], W, n, i, j;
    printf("\n 0/1 Knapsack using Dynamic Programming");
    printf("\n How many objects?-->");
    scanf("%d", &n);
    printf("\n Enter the weights of %d objects:\n", n);
    for(i=0;i<n;i++)
        scanf("%d", &wt[i]);
    printf("\n Enter the values of %d objects:\n", n);
    for(i=0;i<n;i++)
        scanf("%d", &val[i]);
    printf("\n Enter the maximum capacity of the Knapsack-->");
    scanf("%d", &W);
    knapSack(W, wt, val, n);
    return 0;
}
```

Sample output:

```

0/1 Knapsack using Dynamic Programming
How many objects?-->4

Enter the weights of 4 objects:
1 2 4 2

Enter the values of 4 objects:
5 3 5 3

Enter the maximum capacity of the Knapsack-->4

The solution table:
0 0 0 0 0
0 5 5 5 5
0 5 5 8 8
0 5 5 8 8
0 5 5 8 8

The Selected Items: Item-2 Item-1

The Maximum profit gained = 8

```

Outcome of the Exercise:

The program successfully solves the 0/1 Knapsack problem using dynamic programming, providing the maximum value that can be obtained and the selection of items to achieve that value within the given weight capacity.

Interview Questions:**1. What is the 0/1 Knapsack problem, and in which scenarios does it find practical application?**

- *Answer:* The 0/1 Knapsack problem involves selecting items with specific weights and values to maximize the total value within a given weight capacity. It finds applications in resource allocation, inventory management, and optimization problems.

2. Explain the concept of dynamic programming and how it is applied to solve the 0/1 Knapsack problem.

- *Answer:* Dynamic programming involves breaking down a complex problem into smaller subproblems and solving them independently. In the case of the 0/1 Knapsack problem, a table is filled iteratively to represent optimal solutions to subproblems, leading to the overall solution.

3. How is the dynamic programming table initialized, and what information does it store?

- *Answer:* The table is initialized with zeros. It stores the maximum value that can be obtained for different subproblems, considering the available items and weight capacities.

4. Why is the dynamic programming table filled in a bottom-up manner?

- *Answer:* Filling the table in a bottom-up manner ensures that optimal solutions to subproblems are available when needed, leading to the efficient computation of the overall solution.

5. What is the significance of the decision-making logic in the dynamic programming loop?

- *Answer:* The decision-making logic determines whether to include or exclude the current item in the knapsack. It is based on comparing the total value obtained by including the item and excluding the item, choosing the option with the higher value.

6. How does the program handle the case when the weight of a selected item exceeds the remaining capacity?

- *Answer:* If the weight of the current item exceeds the remaining capacity, the program excludes the item from the knapsack, ensuring that the weight constraint is not violated.

7. Can the 0/1 Knapsack problem be solved using greedy algorithms?

- *Answer:* While greedy algorithms are suitable for some knapsack variations, they may not guarantee an optimal solution for the 0/1 Knapsack problem due to its binary nature.

8. How does the program trace and print the selected items contributing to the maximum value?

- *Answer:* The program traces the items by backtracking through the dynamic programming table, identifying the items that were included in the optimal solution.

9. What modifications would be needed if the weights and values of items were read from an external

	<p>file?</p> <ul style="list-style-type: none"> <i>Answer:</i> The program would need modifications to read the weights and values from an external file, ensuring the data is appropriately processed and utilized in the knapsack calculation. <p>10. In what scenarios would a brute-force approach be impractical for solving the 0/1 Knapsack problem?</p> <ul style="list-style-type: none"> <i>Answer:</i> The 0/1 Knapsack problem involves exponential time complexity when solved using brute-force approaches, making it impractical for larger instances with numerous items. Dynamic programming provides a more efficient solution in such cases.
4	<p>From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.</p> <p>Problem Statement: The task is to find the shortest paths from a given source vertex to all other vertices in a weighted connected graph. The graph can be represented by an adjacency matrix, where each edge has a non-negative weight. Dijkstra's algorithm efficiently computes the shortest paths in such graphs.</p> <p>Solution Overview: Dijkstra's algorithm is used to solve the problem. It is a greedy algorithm that maintains a set of vertices with known minimum distances from the source vertex. The algorithm iteratively selects the vertex with the minimum distance, updates the distances to its neighbors, and continues until all vertices are processed.</p> <p>Intuition: The algorithm starts with the source vertex, initializing distances to all other vertices as infinity. It then explores neighbors of the source vertex, updating their distances based on the weights of the connecting edges. The process repeats, selecting the vertex with the minimum distance in each iteration until all vertices are visited.</p> <p>Code Implementation:</p> <pre>#include<stdio.h> #define SIZE 10 #define INFINITY 999 int cam[SIZE][SIZE],n; void getInput() { int i,j; printf("\n Enter the Cost Adjacency matrix in the matrix form:\n"); for(i=0;i<n;i++) for(j=0;j<n;j++) { scanf("%d",&cam[i][j]); if(cam[i][j]==0) cam[i][j]=INFINITY; } } void displayPath(int dist[],int path[],int src,int dest) { int i=dest; printf("\n"); while(i!=src) { printf("%d<---",i); i=path[i]; } printf("%d, Distance = %d",i,dist[dest]); }</pre>

```
}
void dijkstras(int dist[],int path[],int src, int dest)
{
    int s[SIZE],i,j,u,v,min;
    for(i=0;i<n;i++)
    {
        dist[i]=cam[src][i];
        s[i]=0;
        path[i]=src;
    }
    s[src]=1;
    for(i=1;i<n;i++)
    {
        min=INFINITY;
        u=-1;
        for(j=0;j<n;j++)
        {
            if(dist[j]< min && s[j]==0)
            {
                min=dist[j];
                u=j;
            }
        }
        if(u ==-1) return;
        s[u] = 1;
        if(u == dest) return;
        for(v=0;v<n;v++)
        {
            if(dist[u]+ cam[u][v] < dist[v] && s[v]==0)
            {
                dist[v]=dist[u]+cam[u][v];
                path[v]=u;
            }
        }
    }
}

int main()
{
    int dist[SIZE],path[SIZE],src,i;
    printf("\n Dijkstras Algorithm using Greedy Method");
    printf("\n *****");
    printf("\n Enter the number of vertices in a graph--->");
    scanf("%d",&n);
    getInput();
    printf("\n Enter the source vertex-->");
    scanf("%d",&src);
    printf("\n The shortest paths from %d to all other vertices is:\n",src);
    for(i=0;i<n;i++)
    {
        dijkstras(dist,path,src,i);
        if(src==i)
            printf("\n %d<---%d, Distance=0",src,i);
    }
}
```

```
        else if(dist[i]==INFINITY)
            printf("\n %d is not reachable from %d",i,src);
        else
            displayPath(dist,path,src,i);
    }
    printf("\n");
}
```

Sample Output:

```
Dijkstras Algorithm using Greedy Method
*****
Enter the number of vertices in a graph-->3

Enter the Cost Adjacency matrix in the matrix form:
3 6 2
8 1 0
7 3 4

Enter the source vertex-->0

The shortest paths from 0 to all other vertices is:

0<---0, Distance=0
1<---2<---0, Distance = 5
2<---0, Distance = 2

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully applies Dijkstra's algorithm to find the shortest paths from a given source vertex to all other vertices in a weighted connected graph. It outputs the distances from the source vertex to each destination vertex.

Interview Questions:**1. What is Dijkstra's algorithm, and how does it work?**

- *Answer:* Dijkstra's algorithm is a greedy algorithm that finds the shortest paths from a source vertex to all other vertices in a weighted graph. It iteratively selects the vertex with the minimum distance, updates distances to its neighbors, and repeats until all vertices are processed.

2. Explain the significance of the dist array in the algorithm.

- *Answer:* The **dist** array stores the shortest distances from the source vertex to all other vertices. It is initialized to infinity, and the algorithm updates it during each iteration based on the current minimum distances.

3. How does the program handle vertices with no direct edges from the source vertex?

- *Answer:* The program sets the distance to vertices with no direct edges to infinity initially. If a shorter path is discovered during the algorithm, the distance is updated.

4. What is the purpose of the sptSet array, and how is it utilized in the algorithm?

- *Answer:* The **sptSet** array is used to keep track of visited vertices. It ensures that the algorithm does not revisit vertices that have already been processed, preventing unnecessary calculations.

5. How does the algorithm handle weighted edges with non-negative values?

- *Answer:* The algorithm considers only non-negative weights. It updates the distance to a vertex if a shorter path is found through the current vertex being processed.

6. What modifications would be needed to handle a graph with negative edge weights?

- *Answer:* Dijkstra's algorithm is not suitable for graphs with negative edge weights. For handling negative weights, a different algorithm like Bellman-Ford should be used.

7. What is the time complexity of Dijkstra's algorithm, and in what scenarios is it efficient?

	<ul style="list-style-type: none"> <i>Answer:</i> The time complexity is $O(V^2)$ with an adjacency matrix or $O((V + E) * \log(V))$ with a min-priority queue using an adjacency list. It is efficient for dense graphs and scenarios where edge weights are non-negative. <p>8. Can the algorithm handle graphs with cycles?</p> <ul style="list-style-type: none"> <i>Answer:</i> Dijkstra's algorithm is designed for graphs without cycles. It may produce incorrect results in the presence of cycles, as it assumes that once a vertex is marked as visited, its shortest path is found. <p>9. How does the program ensure that the correct shortest paths are identified during backtracking?</p> <ul style="list-style-type: none"> <i>Answer:</i> The program backtracks by selecting the vertex with the minimum distance in each iteration. This ensures that the shortest paths are considered while updating the distances. <p>10. In what real-world scenarios is Dijkstra's algorithm commonly used?</p> <ul style="list-style-type: none"> <i>Answer:</i> Dijkstra's algorithm is used in network routing, transportation systems, and logistics to find the shortest paths between locations. It also finds applications in robotics and resource management.
5	<p>Find the Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.</p> <p>Problem Statement: The task is to find the Minimum Cost Spanning Tree (MCST) of a given undirected graph. A Minimum Cost Spanning Tree is a subset of the edges of the graph that connects all vertices with the minimum possible total edge weight, without forming cycles. Kruskal's algorithm is a greedy approach to solving this problem.</p> <p>Solution Overview: Kruskal's algorithm works by sorting all the edges based on their weights and then selecting edges in ascending order while ensuring that adding each edge does not create a cycle in the evolving tree. The process continues until all vertices are connected.</p> <p>Intuition:</p> <ol style="list-style-type: none"> Sort all edges in non-decreasing order based on their weights. Initialize an empty tree. Start selecting edges with the smallest weight. Add each selected edge to the tree only if it does not create a cycle. Repeat until all vertices are connected. <p>Code Implementation (in C Programming Language):</p> <pre>#include<stdio.h> #define SIZE 10 #define INFINITY 999 void kruskals(int cam[SIZE][SIZE],int n) { int parent[SIZE],i,j,x,y,u,v,min,mincost,count; for(i=1;i<=n;i++) parent[i]=0; mincost=0; count=1; printf("\n There are %d edges in this graph:\n",n-1); while(count<n) { min=INFINITY; for(i=1;i<=n;i++) for(j=1;j<=n;j++) { if(cam[i][j]<min) { min=cam[i][j]; x=u=i,y=v=j; } } }</pre>

```
    }
    while(parent[u])u=parent[u];
    while(parent[v])v=parent[v];
    if(u!=v)
    {
        printf("\n %d Edge--->(%d,%d), cost=%d",count,x,y,min);
        mincost+=min;
        parent[v]=u;
        count++;
    }
    cam[x][y]=cam[y][x]=INFINITY;
}
printf("\n The minimal cost of the above spanning tree is--->%d",mincost);
}
int main()
{
    int cam[SIZE][SIZE],i,j,n;
    printf("\n Get the cost adjacency matrix ready for ur undirected weighted graph.\n");
    printf("\n Enter how many nodes--->");
    scanf("%d",&n);
    printf("\n Enter the cost adjacency matrix in matrix form:(For no edge enter 0):\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cam[i][j]);
            if(cam[i][j]==0)
                cam[i][j]=INFINITY;
        }
    printf("\n The minimum spanning tree for the given graph is:\n");
    kruskals(cam,n);
    printf("\n");
    return(0);
}
```

Get the cost adjacency matrix ready for ur undirected weighted graph.

Enter how many nodes--->6

Enter the cost adjacency matrix in matrix form:(For no edge enter 0):

```
0 3 0 0 6 5
3 0 1 0 0 4
0 1 0 6 0 4
0 0 6 0 8 5
6 0 0 8 0 2
5 4 4 5 2 0
```

The minimum spanning tree for the given graph is:

There are 5 edges in this graph:

```
1 Edge--->(2,3), cost=1
2 Edge--->(5,6), cost=2
3 Edge--->(1,2), cost=3
4 Edge--->(2,6), cost=4
5 Edge--->(4,6), cost=5
```

The minimal cost of the above spanning tree is--->15

Outcome of the Exercise:

The program successfully applies Kruskal's algorithm to find the Minimum Cost Spanning Tree of the given undirected graph. It outputs the edges that form the Minimum Cost Spanning Tree.

Interview Questions:**1. What is a Minimum Cost Spanning Tree, and in what practical scenarios is it useful?**

- *Answer:* A Minimum Cost Spanning Tree is a subset of edges of a connected, undirected graph with the minimum possible total edge weight. It finds applications in network design, circuit design, and transportation planning.

2. How does Kruskal's algorithm work to find the Minimum Cost Spanning Tree?

- *Answer:* Kruskal's algorithm works by sorting edges based on weights and then iteratively selecting edges in ascending order while ensuring that each selected edge does not form a cycle in the evolving tree.

3. What is the role of the parent array in Kruskal's algorithm?

- *Answer:* The **parent** array is used for disjoint-set data structure operations. It keeps track of the sets to which each vertex belongs, allowing the algorithm to determine whether adding an edge would create a cycle.

4. How does the algorithm ensure that the Minimum Cost Spanning Tree does not contain cycles?

- *Answer:* The algorithm uses the disjoint-set data structure to keep track of sets of vertices. It checks whether adding an edge between two vertices would create a cycle by verifying if they already belong to the same set.

5. Why does the algorithm start by sorting edges in non-decreasing order based on weights?

- *Answer:* Sorting edges based on weights allows the algorithm to consider edges in ascending order, ensuring that the smallest-weighted edges are added first and preventing the formation of cycles.

6. What is the time complexity of Kruskal's algorithm, and in what scenarios is it efficient?

- *Answer:* The time complexity is $O(E \log E)$ for sorting edges, where E is the number of edges. It is efficient for sparse graphs where the number of edges is significantly less than the number of vertices.

7. Can Kruskal's algorithm handle graphs with negative edge weights?

- *Answer:* Yes, Kruskal's algorithm can handle graphs with negative edge weights as long as there are no negative cycles. However, in practice, other algorithms like Prim's algorithm or Boruvka's algorithm might be preferred for graphs with negative weights.

8. What modifications would be needed if the graph edges were read from an external file?

- *Answer:* The program would need modifications to read edge information from an external file, ensuring proper parsing and conversion of data.

9. How does the program output the edges that form the Minimum Cost Spanning Tree?

- *Answer:* The program outputs the selected edges during the execution of Kruskal's algorithm, indicating which edges are added to the Minimum Cost Spanning Tree.

10. What happens if the input graph is not connected or has disconnected components?

- *Answer:* Kruskal's algorithm assumes a connected graph. If the graph is not connected, the algorithm will find a Minimum Cost Spanning Forest, which is a collection of Minimum Cost Spanning Trees for each connected component.

6

Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.**Problem Statement:**

The task is to find the Minimum Cost Spanning Tree (MCST) of a given undirected graph using Prim's algorithm. A Minimum Cost Spanning Tree is a subset of the edges of the graph that connects all vertices with the minimum possible total edge weight, without forming cycles.

Solution Overview:

Prim's algorithm is a greedy algorithm that starts with an arbitrary vertex and incrementally grows the Minimum Cost Spanning Tree by adding the edge with the smallest weight that connects a vertex in the tree to a vertex outside the tree. This process continues until all vertices are included in the tree.

Intuition:

1. Start with an arbitrary vertex as the initial tree.
2. Grow the tree by adding the minimum-weight edge that connects a vertex in the tree to a vertex outside the tree.
3. Repeat until all vertices are included in the tree.

Code Implementation:

```
#include<stdio.h>
#define SIZE 10
#define INFINITY 999

void prims(int cam[SIZE][SIZE],int n)
{
    int visited[SIZE],i,j,u,v,min,mincost,count;
    visited[0]=1;
    for(i=1;i<n;i++)
        visited[i]=0;
    mincost=0;
    count=1;
    printf("\n There are %d edges in this graph:\n",n-1);
    while(count<n)
    {
        min=INFINITY;
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
            {
                if(cam[i][j]<min && visited[i]==1)
                {
                    min=cam[i][j];
                    u=i,v=j;
                }
            }
        if(visited[v]==0)
        {
            printf("\n %d Edge--->(%d,%d), cost=%d",count,u,v,min);
            mincost+=min;
            visited[v]=1;
            count++;
        }
        cam[u][v]=cam[v][u]=INFINITY;
    }
    printf("\n The minimal cost of the above spanning tree is--->%d",mincost);
}

int main()
{
    int cam[SIZE][SIZE],i,j,n;
    printf("\n Prims Algorithm to find MCSP of a weighted graph");
    printf("\n *****");
    printf("\n Get the cost adjacency matrix ready for your undirected weighted graph.\n");
    printf("\n Enter how many vertices--->");
    scanf("%d",&n);
    printf("\n Enter the cost adjacency matrix in matrix form:(For no edge enter 0):\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
```

```

scanf("%d",&cam[i][j]);

if(cam[i][j]==0)
    cam[i][j]=INFINITY;
}
}

printf("\n The minimum spanning tree for the given graph is:\n");
prims(cam,n);
printf("\n");
return(0);
}

```

Sample output:

```

Prims Algorithm to find MCSP of a weighted graph
*****
Get the cost adjacency matrix ready for your undirected weighted graph.

Enter how many vertices--->6

Enter the cost adjacency matrix in matrix form:(For no edge enter 0):
0 3 0 0 6 5
3 0 1 0 0 4
0 1 0 6 0 4
0 0 6 0 8 5
6 0 0 8 0 2
5 4 4 5 2 0

The minimum spanning tree for the given graph is:

There are 5 edges in this graph:

1 Edge--->(0,1), cost=3
2 Edge--->(1,2), cost=1
3 Edge--->(1,5), cost=4
4 Edge--->(5,4), cost=2
5 Edge--->(5,3), cost=5
The minimal cost of the above spanning tree is--->15

```

Outcome of the Exercise:

The program successfully applies Prim's algorithm to find the Minimum Cost Spanning Tree of the given undirected graph. It outputs the edges that form the Minimum Cost Spanning Tree.

Interview Questions:**1. What is Prim's algorithm, and how does it work?**

- *Answer:* Prim's algorithm is a greedy algorithm that finds the Minimum Cost Spanning Tree of a connected, undirected graph. It starts with an arbitrary vertex and grows the tree by adding the minimum-weight edge that connects a vertex in the tree to a vertex outside the tree.

2. How does the program initialize key values and the set in Prim's algorithm?

- *Answer:* Key values are initialized to infinity, and the set is initialized to include no vertices. These data structures are used to keep track of the minimum weight edge and vertices included in the MST.

3. Explain the role of the minKey function in Prim's algorithm.

- *Answer:* The **minKey** function finds the vertex with the minimum key value among the vertices not yet included in the MST. It is used to select the next vertex to be included in the MST.

4. How is the MST constructed in Prim's algorithm, and what is the termination condition?

- *Answer:* The MST is constructed by iteratively adding the minimum-weight edge that connects a vertex in the tree to a vertex outside the tree. The algorithm terminates when all vertices are included in the MST.

5. What is the significance of the parent array in Prim's algorithm?

- *Answer:* The **parent** array keeps track of the parent of each vertex in the MST. It is used to identify the edges that form the MST during the final output.

6. What happens if the input graph is not connected or has disconnected components in Prim's algorithm?

	<ul style="list-style-type: none"> <i>Answer:</i> Prim's algorithm assumes a connected graph. If the graph is not connected, the algorithm will find the Minimum Cost Spanning Forest, which is a collection of Minimum Cost Spanning Trees for each connected component. <p>7. What is the time complexity of Prim's algorithm, and in what scenarios is it efficient?</p> <ul style="list-style-type: none"> <i>Answer:</i> The time complexity is $O(V^2)$ with an adjacency matrix or $O((V + E) * \log(V))$ with a min-priority queue using an adjacency list. It is efficient for dense graphs and scenarios where edge weights are non-negative. <p>8. How does the program output the edges that form the Minimum Cost Spanning Tree?</p> <ul style="list-style-type: none"> <i>Answer:</i> The program outputs the selected edges during the execution of Prim's algorithm, indicating which edges are added to the Minimum Cost Spanning Tree. <p>9. What modifications would be needed if the graph edges were read from an external file?</p> <ul style="list-style-type: none"> <i>Answer:</i> The program would need modifications to read edge information from an external file, ensuring proper parsing and conversion of data. <p>10. Can Prim's algorithm handle graphs with negative edge weights?</p> <ul style="list-style-type: none"> <i>Answer:</i> Prim's algorithm assumes non-negative edge weights. If the graph has negative edge weights, the algorithm might not provide the correct results. Other algorithms like Kruskal's or Boruvka's may be more suitable for such cases.
7	<p>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.</p> <p>Problem Statement: The task is to implement a scheme to find the optimal solution for the Traveling Salesperson Problem (TSP) and then solve the same problem instance using any approximation algorithm. The goal is to determine the error in the approximation compared to the optimal solution.</p> <p>Solution Overview: The Traveling Salesperson Problem is an NP-hard problem that seeks to find the shortest possible route that visits a set of cities and returns to the starting city. Solving it optimally for large instances can be computationally expensive. One common approach is to use approximation algorithms that provide reasonably good solutions in a shorter amount of time.</p> <p>Intuition:</p> <ol style="list-style-type: none"> Optimal Solution (Exact Algorithm): <ul style="list-style-type: none"> Use an exact algorithm (e.g., dynamic programming) to find the optimal solution for the TSP. Approximation Algorithm: <ul style="list-style-type: none"> Select an approximation algorithm (e.g., nearest neighbor, minimum spanning tree) for a quick but suboptimal solution. Determine Error: <ul style="list-style-type: none"> Compare the cost of the solution obtained from the approximation algorithm with the cost of the optimal solution to calculate the error. <p>Code Implementation:</p> <pre>#include <stdio.h> #define MAX_CITIES 4 #define INF 999 int cam[MAX_CITIES][MAX_CITIES]; int numCities; // Function to find the optimal solution using dynamic programming int optimalTSP(int mask, int pos) { int minCost, city, newCost; if (mask == (1 << numCities) - 1) // If all the cities are visited</pre>

```
        return cam[pos][0];

    minCost = INF;

    for (city = 0; city < numCities; city++)
    {
        if ((mask & (1 << city)) == 0)
        {
            newCost = cam[pos][city] + optimalTSP(mask | (1 << city), city);
            minCost = (newCost < minCost) ? newCost : minCost;
        }
    }

    return minCost;
}

// Function to find the approximate solution using the nearest neighbor algorithm
int approximateTSP()
{
    int i, nearestCity, minCost, nextCity;

    int visited[MAX_CITIES] = {0}; // No city is visited yet
    int currentCity = 0; // Current City is the city with index 0
    int totalCost = 0; // The total cost is initialized to zero

    visited[currentCity] = 1; // Visit the current City

    for (i = 0; i < numCities - 1; i++)
    {
        nearestCity = -1;
        minCost = INF;

        // Find the unvisited nearest city from the current city
        for (nextCity = 0; nextCity < numCities; nextCity++)
        {
            if (!visited[nextCity] && cam[currentCity][nextCity] < minCost)
            {
                nearestCity = nextCity;
                minCost = cam[currentCity][nextCity];
            }
        }

        visited[nearestCity] = 1; // Visit the nearest City
        totalCost += minCost; // Accumulate the cost
        currentCity = nearestCity; // Make the nearest city as the new current city and loop through
    }

    totalCost += cam[currentCity][0]; // Find the total cost

    return totalCost;
}
```

```
int main()
{
    int i, j, optimalCost, approximateCost, error;

    printf("\n TSP with Dynamic Programming and Nearest Neighbor Algorithm");
    printf("\n *****");
    printf("\n Get the cost adjacency matrix ready for a weighted graph.\n");

    printf("\n Enter how many cities?(No more than 4 Cities)--->");
    scanf("%d",&numCities);

    printf("\n Enter the cost adjacency matrix in matrix form:(For no edge enter 0):\n");
    for(i=0;i<numCities;i++)
    {
        for(j=0;j<numCities;j++)
        {
            scanf("%d",&cam[i][j]);
            if(i!=j && cam[i][j]==0)
                cam[i][j]=INF;
        }
    }

    optimalCost = optimalTSP(1, 0);
    approximateCost = approximateTSP();
    error = approximateCost - optimalCost;

    printf("Optimal TSP Cost: %d\n", optimalCost);
    printf("Approximate TSP Cost: %d\n", approximateCost);
    printf("Error in Approximation: %d\n", error);

    return 0;
}
```

Sample output:

```
TSP with Dynamic Programming and Nearest Neighbor Algorithm
*****
Get the cost adjacency matrix ready for a weighted graph.

Enter how many cities?(No more than 4 Cities)--->4

Enter the cost adjacency matrix in matrix form:(For no edge enter 0):
0 2 4 8
2 0 7 5
4 7 0 3
8 5 3 0
Optimal TSP Cost: 14
Approximate TSP Cost: 14
Error in Approximation: 0

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program demonstrates finding the optimal solution for the Traveling Salesperson Problem using dynamic programming and then uses the nearest neighbor algorithm as an approximation. It calculates and outputs the error in the approximation compared to the optimal solution.

Interview Questions:**1. What is the Traveling Salesperson Problem, and why is it considered a challenging problem?**

- *Answer:* The Traveling Salesperson Problem involves finding the shortest possible route that visits a set of cities and returns to the starting city. It is challenging because it is NP-hard, meaning there is no known polynomial-time algorithm to solve it optimally for all instances.

2. Explain the intuition behind the dynamic programming approach to solving the TSP.

- *Answer:* The dynamic programming approach systematically explores all possible combinations of cities and their orders, calculating the total cost for each combination. The optimal solution is obtained by selecting the combination with the minimum cost.

3. Why is dynamic programming used to find the optimal solution rather than a greedy approach?

- *Answer:* Dynamic programming ensures optimality by considering all possible combinations and avoiding suboptimal choices. A greedy approach, while faster, may not guarantee the optimal solution.

4. What is the nearest neighbor algorithm, and how does it work in approximating the TSP?

- *Answer:* The nearest neighbor algorithm starts from an arbitrary city and repeatedly selects the nearest unvisited city until all cities are visited. It provides a quick but suboptimal solution by making locally optimal choices.

5. How does the program calculate the error in the approximation?

- *Answer:* The error in the approximation is calculated by subtracting the cost of the optimal solution from the cost of the approximate solution.

6. What are the advantages and disadvantages of using approximation algorithms for the TSP?

- *Answer:* Approximation algorithms are faster but may not guarantee optimality. They are suitable for large instances where finding the optimal solution is impractical.

7. Can you suggest other approximation algorithms commonly used for the TSP?

- *Answer:* Other approximation algorithms include the Christofides algorithm, the Lin-Kernighan algorithm, and the Ant Colony Optimization algorithm.

8. How does the error in the approximation change with the size of the TSP instance?

- *Answer:* In general, the error in the approximation tends to increase as the size of the TSP instance grows. Approximation algorithms may provide better results for smaller instances compared to larger ones.

9. What modifications would be needed if the TSP instance were read from an external file?

- *Answer:* The program would need modifications to read the TSP instance (e.g., adjacency matrix or distance matrix) from an external file, ensuring proper parsing and conversion of data.

8

a) For a given graph, print all the nodes reachable from a given starting node in a digraph using Breadth First Search method.

Problem Statement:

The task is to print all the nodes reachable from a given starting node in a directed graph using the Breadth-First Search (BFS) method. The goal is to systematically explore all reachable nodes in a breadthward motion from the starting node.

Solution Overview:

Breadth-First Search is a graph traversal algorithm that explores all the vertices at the current level before moving on to the vertices at the next level. It is particularly useful for finding the shortest path in an unweighted graph and can be applied to discover all reachable nodes from a given starting node.

Intuition:**1. Queue-based Exploration:**

- Use a queue data structure to keep track of the nodes to be visited.
- Enqueue the starting node.
- While the queue is not empty, dequeue a node, print it, and enqueue its unvisited neighbors.

2. Mark Visited Nodes:

- Use a boolean array to mark nodes as visited to avoid redundant processing.

Code Implementation:

```
#include <stdio.h>

#define MAX_NODES 10

// Function to perform Breadth First Search
void BFS(int graph[MAX_NODES][MAX_NODES], int numNodes, int startNode)
{
    int visited[MAX_NODES] = {0};
    int queue[MAX_NODES];
    int front = 0, rear = 0, currentNode, i;

    visited[startNode] = 1;
    queue[rear++] = startNode;

    while (front < rear)
    {
        currentNode = queue[front++];

        printf("%d ", currentNode); // Print the current node

        // Explore all adjacent nodes of the current node
        for (i = 0; i < numNodes; i++)
        {
            if (graph[currentNode][i] && !visited[i])
            {
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
}

int main()
{
    int graph[MAX_NODES][MAX_NODES] = {0};
    int numNodes, startNode, i, j;

    printf("Breadth First Search Graph Traversal:\n");
    printf("*****\n");
    printf("Enter the number of nodes(vertices) of a digraph: ");
    scanf("%d", &numNodes);

    printf("Enter the adjacency matrix (1 if there is an edge, 0 otherwise):\n");
    for (i = 0; i < numNodes; i++)
    {
        for (j = 0; j < numNodes; j++)
        {
            scanf("%d", &graph[i][j]);
        }
    }
}
```

```
}

printf("Enter the starting node: ");
scanf("%d", &startNode);

printf("Nodes reachable from node %d using BFS: ", startNode);
BFS(graph, numNodes, startNode);
printf("\n");

return 0;
}
```

Sample Output:

```
Breadth First Search Graph Traversal:
*****
Enter the number of nodes(vertices) of a digraph: 5
Enter the adjacency matrix (1 if there is an edge, 0 otherwise):
0 1 1 0 1
1 0 1 0 0
1 1 0 1 0
0 0 1 0 0
1 0 0 0 0
Enter the starting node: 0
Nodes reachable from node 0 using BFS: 0 1 2 4 3

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully prints all nodes reachable from a given starting node in a directed graph using the Breadth-First Search method. It systematically explores the graph in breadthward motion, avoiding redundant processing of already visited nodes.

Interview Questions:

- 1. What is Breadth-First Search (BFS), and how does it differ from Depth-First Search (DFS)?**
 - *Answer:* BFS is a graph traversal algorithm that explores all vertices at the current level before moving on to the next level. DFS explores as far as possible along each branch before backtracking.
- 2. Explain the role of the queue data structure in BFS.**
 - *Answer:* The queue is used to keep track of the nodes to be visited in a first-in, first-out manner. Nodes are enqueued when discovered and dequeued when processed.
- 3. How does the program avoid redundant processing of nodes in BFS?**
 - *Answer:* The program uses a boolean array (**visited**) to mark nodes as visited. Before enqueueing a neighbor, it checks whether the neighbor is already marked as visited.
- 4. What is the significance of the adjacency matrix in representing a directed graph?**
 - *Answer:* The adjacency matrix indicates the presence or absence of directed edges between nodes. In this program, a value of 1 in **adjacencyMatrix[i][j]** signifies a directed edge from node **i** to node **j**.
- 5. How would the program change if it needed to handle a weighted directed graph?**
 - *Answer:* For a weighted graph, the adjacency matrix would store weights instead of binary values. The traversal logic would remain similar, but edge weights would be considered in specific scenarios.
- 6. What modifications would be needed if the graph edges were read from an external file?**
 - *Answer:* The program would need modifications to read edge information from an external file, ensuring proper parsing and conversion of data.
- 7. Can BFS be applied to an undirected graph, and how would it differ?**
 - *Answer:* Yes, BFS can be applied to an undirected graph. The primary difference is that in an undirected graph, there is no concept of incoming or outgoing edges. The adjacency matrix would be symmetric.

	<p>8. What is the time complexity of BFS, and under what scenarios is it efficient?</p> <ul style="list-style-type: none"> <i>Answer:</i> The time complexity of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges. It is efficient for finding the shortest path in an unweighted graph or determining connectivity.
8	<p>b) For a given graph, check whether a given graph is connected or not using DFS method.</p> <p>Problem Statement: The task is to check whether a given graph is connected or not using the Depth-First Search (DFS) method. The goal is to determine if there exists a path between every pair of nodes in the graph.</p> <p>Solution Overview: Depth-First Search is a graph traversal algorithm that explores as far as possible along each branch before backtracking. By applying DFS and visiting all nodes, we can determine if the graph is connected.</p> <p>Intuition:</p> <p>DFS Exploration:</p> <ul style="list-style-type: none"> Start DFS from an arbitrary node. Mark each visited node. If all nodes are visited, the graph is connected. <p>Code Implementation:</p> <pre>#include <stdio.h> #define MAX_NODES 10 // Function to perform Depth First Search void DFS(int graph[MAX_NODES][MAX_NODES], int numNodes, int currentNode, int visited[]) { int i; visited[currentNode] = 1; // Explore all adjacent nodes of the current node for (i = 0; i < numNodes; i++) { if (graph[currentNode][i] && !visited[i]) { DFS(graph, numNodes, i, visited); } } } // Function to check if the graph is connected int isConnected(int graph[MAX_NODES][MAX_NODES], int numNodes) { int visited[MAX_NODES] = {0}; int i; // Perform DFS from the first node DFS(graph, numNodes, 0, visited); // Check if all nodes are visited for (i = 0; i < numNodes; i++) { if (!visited[i])</pre>

```
{
    return 0;
}

return 1;
}

int main()
{
    int graph[MAX_NODES][MAX_NODES] = {0};
    int numNodes, i,j;

    printf("Depth First Search Graph Traversal:\n");
    printf("*****\n");
    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);

    printf("Enter the adjacency matrix (1 if there is an edge, 0 otherwise):\n");
    for (i = 0; i < numNodes; i++)
    {
        for (j = 0; j < numNodes; j++)
        {
            scanf("%d", &graph[i][j]);
        }
    }

    if (isConnected(graph, numNodes))
    {
        printf("The graph is connected.\n");
    }
    else
    {
        printf("The graph is not connected.\n");
    }

    return 0;
}
```

Sample Output:

```
Depth First Search Graph Traversal:
*****
Enter the number of nodes: 5
Enter the adjacency matrix (1 if there is an edge, 0 otherwise):
0 1 1 0 1
1 0 1 0 0
1 1 0 1 0
0 0 1 0 0
1 0 0 0 0
The graph is connected.

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully checks whether a given graph is connected or not using the Depth-First Search method. It explores the graph, marks visited nodes, and determines if all nodes are visited, indicating connectivity.

Interview Questions:**1. What is the significance of the DFS algorithm in checking graph connectivity?**

- *Answer:* DFS explores the graph and marks visited nodes, allowing us to check if there exists a path between every pair of nodes. If all nodes are visited during DFS, the graph is connected.

2. How does the program ensure all nodes are visited during DFS?

- *Answer:* The program uses a boolean array (**visited**) to mark nodes as visited. After DFS, it checks if all nodes are marked as visited to determine graph connectivity.

3. Can DFS be applied to an undirected graph for checking connectivity, and how would it differ?

- *Answer:* Yes, DFS can be applied to an undirected graph for checking connectivity. The primary difference is that in an undirected graph, there is no concept of incoming or outgoing edges.

4. Explain the role of the adjacency matrix in representing a directed graph.

- *Answer:* The adjacency matrix indicates the presence or absence of directed edges between nodes. In this program, a value of 1 in **adjacencyMatrix[i][j]** signifies a directed edge from node **i** to node **j**.

5. How does the program handle disconnected components in a graph?

- *Answer:* The program starts DFS from an arbitrary node and checks if all nodes are visited. If the graph has disconnected components, not all nodes will be visited, indicating that the graph is not connected.

6. What modifications would be needed if the graph edges were read from an external file?

- *Answer:* The program would need modifications to read edge information from an external file, ensuring proper parsing and conversion of data.

7. What is the time complexity of DFS, and in what scenarios is it efficient?

- *Answer:* The time complexity of DFS is $O(V + E)$, where V is the number of vertices and E is the number of edges. It is efficient for checking connectivity and exploring paths in a graph.

9

Implement N Queen's problem using Back Tracking**Problem Statement:**

The N Queens problem involves placing N chess queens on an $N \times N$ chessboard in such a way that no two queens threaten each other. This means that no two queens can be in the same row, column, or diagonal. The goal is to find a placement of queens that satisfies these constraints.

Solution Overview:

The Backtracking algorithm is a suitable approach for solving the N Queens problem. Backtracking systematically explores potential solutions and abandons a partial solution ("backtracks") as soon as it determines that the solution cannot be extended to a valid one.

Intuition:**1. Recursion and Decision Points:**

- Use recursive calls to explore possible configurations.
- At each decision point, try placing a queen in an unoccupied position.

2. Check Constraints:

- Check if placing a queen at the current position violates any constraints.
- If valid, proceed with the next row; otherwise, backtrack.

Code Implementation:

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

```
int board[8][8];
```

```
// Function to print the chessboard
```

```
void printBoard(int n)
```

```
{
```

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        printf("%d ", board[i][j]);
    }
    printf("\n");
}

// Function to check if placing a queen at board[row][col] is safe
int isSafe(int row, int col, int n)
{
    int i,j;
    // Check left side of the current row
    for (i = 0; i < col; i++)
    {
        if (board[row][i] == 1)
        {
            return 0;
        }
    }

    // Check upper diagonal on the left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j] == 1)
        {
            return 0;
        }
    }

    // Check lower diagonal on the left side
    for (i = row, j = col; i < n && j >= 0; i++, j--)
    {
        if (board[i][j] == 1)
        {
            return 0;
        }
    }

    return 1;
}

// Function to solve the N Queens problem using backtracking
int solveNQueens(int col, int n)
{
    int i;
    // Base case: All queens are placed
    if (col == n)
    {
        return 1;
    }
}
```

```
}

for (i = 0; i < n; i++)
{
    // Check if placing a queen at board[i][col] is safe
    if (isSafe(i, col, n))
    {
        // Place the queen
        board[i][col] = 1;

        // Recur to place queens in the remaining columns
        if (solveNQueens(col + 1, n))
        {
            return 1; // Solution found
        }

        // If placing a queen does not lead to a solution, backtrack
        board[i][col] = 0;
    }
}

return 0; // No solution in this branch
}

int main()
{
    int i, j, n;
    printf("\n N-Queens");
    printf("\n *****");

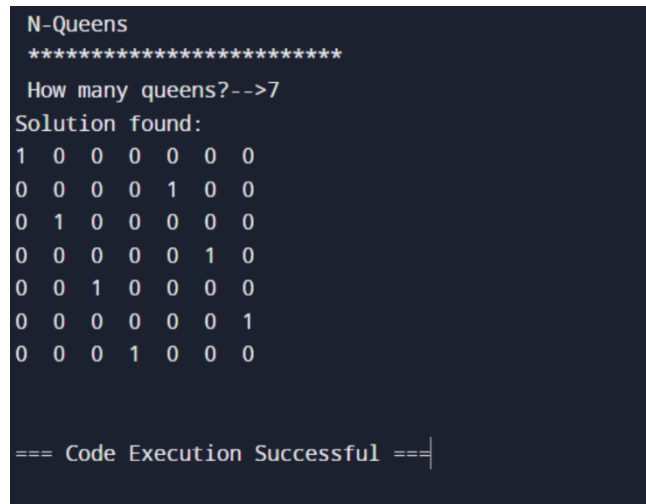
    do
    {
        printf("\n How many queens?-->");
        scanf("%d", &n);
        if (n < 4 || n > 8)
            printf("\n Enter the number of queens(>=4 and <=8");
    } while (n < 4 || n > 8);

    // Initialize the chessboard
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            board[i][j] = 0;
        }
    }

    // Solve the N Queens problem
    if (solveNQueens(0, n))
    {
        printf("Solution found:\n");
        printBoard(n);
    }
}
```

```
} else
{
    printf("No solution exists.\n");
}

return 0;
}
```

Sample Output:

```
N-Queens
*****
How many queens?-->7
Solution found:
1 0 0 0 0 0 0
0 0 0 0 1 0 0
0 1 0 0 0 0 0
0 0 0 0 0 1 0
0 0 1 0 0 0 0
0 0 0 0 0 0 1
0 0 0 1 0 0 0

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully solves the N Queens problem using the Backtracking algorithm. It finds a valid placement of queens on the chessboard such that no two queens threaten each other.

Interview Questions:**1. Explain the N Queens problem and its significance in computer science.**

- *Answer:* The N Queens problem involves placing N queens on an N×N chessboard without any two queens threatening each other. It is a classic problem used to demonstrate problem-solving and algorithmic techniques.

2. Why is the Backtracking algorithm suitable for solving the N Queens problem?

- *Answer:* Backtracking systematically explores potential solutions and abandons a partial solution ("backtracks") as soon as it determines that the solution cannot be extended to a valid one. This aligns well with the N Queens problem's constraints.

3. What are the constraints that a valid placement of queens must satisfy in the N Queens problem?

- *Answer:* No two queens can be in the same row, column, or diagonal. Placing a queen must not threaten any other queens on the chessboard.

4. Explain the role of the isSafe function in the program.

- *Answer:* The **isSafe** function checks if placing a queen at a specific position violates any constraints. It ensures that no queens in the same row, column, or diagonal can threaten each other.

5. How does the program handle the case where no solution exists for the N Queens problem?

- *Answer:* If the **solveNQueens** function returns **false**, the program prints "No solution exists." This indicates that no valid placement of queens on the chessboard satisfies the constraints.

6. What modifications would be needed to extend the program to handle an N×M chessboard?

- *Answer:* The program would need adjustments to handle an N×M chessboard. The **N** constant would be replaced with variables **N** and **M** throughout the code.

7. What is the time complexity of the Backtracking solution for the N Queens problem?

- *Answer:* The time complexity is exponential, $O(N!)$, where **N** is the size of the chessboard. This is because each queen placement involves exploring a branch of possibilities, leading to a combinatorial explosion.

10

Implement All-Pairs Shortest Paths Problem using Floyd's algorithm**Problem Statement:**

The All-Pairs Shortest Paths problem involves finding the shortest paths between every pair of vertices in a weighted directed graph. The goal is to compute a matrix of shortest path distances between all pairs of vertices.

Solution Overview:

Floyd's algorithm is a dynamic programming approach that efficiently solves the All-Pairs Shortest Paths problem. It iteratively updates the shortest path distances between pairs of vertices by considering all intermediate vertices.

Intuition:**1. Dynamic Programming Table:**

- Use a 2D matrix to represent the shortest path distances.
- Initialize the matrix with direct edge weights.

2. Iterative Updates:

- For each intermediate vertex, check if going through it improves the distance between two vertices.
- Update the matrix with the minimum distance.

Code Implementation:

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

```
#define INF 999
```

```
#define V 8
```

```
// Function to print the shortest path matrix
```

```
void printSolution(int dist[][V], int n)
```

```
{
    int i,j;
    printf("Shortest path matrix:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (dist[i][j] == INF)
            {
                printf(" INF ");
            }
            else
            {
                printf("%4d ", dist[i][j]);
            }
        }
        printf("\n");
    }
}
```

```
// Function to solve the All-Pairs Shortest Paths problem using Floyd's algorithm
```

```
void floyd(int cam[][V], int n)
```

```
{
    int dist[V][V];
```

```
// Initialize the distance matrix with direct edge weights
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        dist[i][j] = cam[i][j];
    }
}

// Consider each vertex as an intermediate vertex and update the distance matrix
for (int k = 0; k < n; k++)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (dist[i][k] + dist[k][j] < dist[i][j])
            {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}

// Print the shortest path matrix
printSolution(dist,n);
}

int main()
{
    int cam[V][V],i,j,n;

    printf("\n Floyd's Algorithm to find all pairs shortest paths of a graph");
    printf("\n *****");
    printf("\n How many vertices in your directed weighted graph?-->");
    scanf("%d",&n);
    printf("\n Enter the Cost Adjacency matrix of your graph in the matrix form:\n");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
            scanf("%d",&cam[i][j]);
            if(cam[i][j]==0)
                cam[i][j]=INF;
        }

    // Solve the All-Pairs Shortest Paths problem using Floyd's algorithm
    floyd(cam,n);

    return 0;
}
```


Sample Output:

```
Floyds Algorithm to find all pairs shortest paths of a graph
*****
How many vertices in your directed weighted graph?-->4

Enter the Cost Adjacency matrix of your graph in the matrix form:
0 0 3 0
2 0 0 0
0 7 0 1
6 0 0 0
Shortest path matrix:
10 10 3 4
2 12 5 6
7 7 10 1
6 16 9 10

=== Code Execution Successful ===
```

Outcome of the Exercise:

The program successfully solves the All-Pairs Shortest Paths problem using Floyd's algorithm. It computes and prints the shortest path matrix for a given weighted directed graph.

Interview Questions:**1. Explain the All-Pairs Shortest Paths problem and its applications.**

- *Answer:* The All-Pairs Shortest Paths problem involves finding the shortest paths between every pair of vertices in a weighted directed graph. Applications include network routing, transportation planning, and optimization problems.

2. Why is Floyd's algorithm considered a dynamic programming approach?

- *Answer:* Floyd's algorithm optimally solves subproblems and builds up the solution using the results of smaller subproblems. It iteratively updates the shortest path distances between pairs of vertices, demonstrating the principles of dynamic programming.

3. What is the significance of the INF constant in the program?

- *Answer:* The **INF** constant represents infinity and is used to initialize distances in the matrix. It signifies that there is no direct edge between two vertices.

4. How does the program handle the case where there is no direct edge between two vertices?

- *Answer:* If there is no direct edge between two vertices, the program represents the distance as **INF** in the shortest path matrix.

5. Explain the role of the floydWarshall function in the program.

- *Answer:* The **floydWarshall** function applies Floyd's algorithm to solve the All-Pairs Shortest Paths problem. It iteratively updates the shortest path distances in the matrix, considering each vertex as an intermediate vertex.

6. What modifications would be needed to extend the program for a larger graph?

- *Answer:* The program is scalable for larger graphs. To handle a graph with more vertices, increase the value of the constant **V** and provide the corresponding adjacency matrix.

7. What is the time complexity of Floyd's algorithm, and under what scenarios is it efficient?

- *Answer:* The time complexity is $O(V^3)$, where V is the number of vertices. It is efficient for dense graphs and graphs with a relatively small number of vertices.

PART – B

ALGORITHMS LAB PROJECT

REPORT FORMAT

7. Mini-Project Specification and Evaluation Rubrics

Part-B**1. Mini Projects on Sorting Algorithm Efficiency Comparison**

Overview of the Solution:

The goal of this mini-project is to compare the efficiency of different sorting algorithms. Students are expected to implement and analyze the performance of algorithms like Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. The project involves generating random datasets of varying sizes, applying each sorting algorithm to these datasets, and measuring the time complexity. Students will need to present their findings, highlighting the strengths and weaknesses of each algorithm.

Mini-project Evaluation Rubrics:

1. Implementation (40 points):

- Properly implemented and functional sorting algorithms.
- Code readability and adherence to coding standards.
- Handling of edge cases and error scenarios.

2. Efficiency Analysis (30 points):

- Accurate measurement of time complexity for each sorting algorithm.
- Clear presentation of experimental results through tables or graphs.
- Insightful comparison of the algorithms' performance.

3. Documentation (20 points):

- Well-documented code with comments explaining key sections.
- A detailed report outlining the project objectives, methodology, and results.
- Proper citation of sources and references, if any.

4. Presentation (10 points):

- Clear and engaging oral presentation.
- Effective use of visual aids to convey information.
- Ability to answer questions and defend the chosen approach.

2. Mini Projects on Dynamic Programming

Overview of the Solution:

This mini-project focuses on solving problems using Dynamic Programming (DP) techniques. Students will choose problems that exhibit overlapping subproblems and optimal substructure. Examples include the 0/1 Knapsack problem, Longest Common Subsequence, and Matrix Chain Multiplication. The project involves implementing DP solutions, analyzing time and space complexities, and presenting the optimized solutions.

Mini-project Evaluation Rubrics:

1. Problem Selection (20 points):

- Careful selection of problems showcasing overlapping subproblems and optimal substructure.
- Problems aligning with the scope and complexity of the mini-project.

2. Dynamic Programming Solutions (40 points):

- Correct and efficient implementation of DP solutions.
- Proper utilization of memoization or tabulation techniques.

- Handling edge cases and input variations.
3. **Analysis and Optimization (25 points):**
 - In-depth analysis of time and space complexities.
 - Comparison with alternative approaches and justifications for chosen solutions.
 - Demonstrated understanding of DP principles.
 4. **Documentation (10 points):**
 - Well-documented code with clear explanations of DP strategies.
 - A comprehensive report detailing problem statements, solutions, and findings.
 - Proper citation of sources and references, if any.
 5. **Presentation (5 points):**
 - Concise and effective oral presentation.
 - Clarity in explaining DP concepts and solutions.
 - Ability to address questions and provide insights.
-

3. Mini Projects on Pathfinding

Overview of the Solution:

In this mini-project, students will explore various pathfinding algorithms applied to graphs or grids. Problems may involve finding the shortest path between two points, maze solving, or routing in a network. Common algorithms to be considered are Dijkstra's Algorithm, A* Algorithm, and Depth-First Search. Students will implement these algorithms, analyze their performance, and present their findings.

Mini-project Evaluation Rubrics:

1. **Problem Definition (15 points):**
 - Clearly defined pathfinding problems with appropriate complexity.
 - Problems that require the application of multiple algorithms for comparison.
 2. **Algorithm Implementation (40 points):**
 - Correct and functional implementation of selected pathfinding algorithms.
 - Handling of different grid/graph representations and scenarios.
 3. **Performance Analysis (25 points):**
 - Accurate measurement of algorithmic performance in terms of time and space complexity.
 - Comparative analysis of different algorithms applied to the chosen problems.
 - Identification of strengths and weaknesses of each algorithm.
 4. **Documentation (15 points):**
 - Well-documented code with comments explaining key sections.
 - A comprehensive report outlining problem statements, algorithm implementations, and analysis.
 - Proper citation of sources and references, if any.
 5. **Presentation (5 points):**
 - Clear and engaging oral presentation.
 - Effective use of visual aids to illustrate pathfinding algorithms.
 - Ability to answer questions and discuss insights.
-

4. Mini Projects on Graph Traversal Techniques

Overview of the Solution:

This mini-project focuses on exploring and implementing various graph traversal techniques. Students will choose problems that require Breadth-First Search (BFS), Depth-First Search (DFS), or both. Examples include connectivity analysis, cycle detection, and node reachability. The project involves implementing traversal algorithms, analyzing their applications, and presenting findings.

Mini-project Evaluation Rubrics:

1. Problem Selection (15 points):

- Well-chosen problems that necessitate the application of BFS and/or DFS.
- Problems aligning with the scope and complexity of the mini-project.

2. Traversal Algorithm Implementation (40 points):

- Correct and efficient implementation of BFS and/or DFS.
- Proper handling of graph representations (adjacency matrix, adjacency list, etc.).
- Handling of edge cases and input variations.

3. Applications and Analysis (25 points):

- Identification and demonstration of graph traversal applications in the chosen problems.
- Comparative analysis of BFS and DFS in terms of advantages and disadvantages.
- Insightful discussion on how traversal techniques solve specific problems.

4. Documentation (15 points):

- Well-documented code with comments explaining traversal algorithms.
- A comprehensive report detailing problem statements, traversal implementations, and analysis.
- Proper citation of sources and references, if any.

5. Presentation (5 points):

- Clear and engaging oral presentation.
- Effective use of visual aids to illustrate traversal algorithms.
- Ability to answer questions and discuss insights.

Note:

The students are expected to implement all the 5 Mini-projects. But, present and submit the project report of any one of the five projects, based on the approval of the Guide (Faculty Member Assigned to a student).

8. Project Report Format:**School of Computing Science and Engineering**

Course Code:	Algorithms Lab Project Report	Academic Year:
		Semester & Batch:
Project Details:		
Project Title:		
Place of Project:	REVA UNIVERSITY, BENGALURU	
Student Details:		
Name:		Sign:
Mobile No:		
Email-ID:		
SRN:		
Guide and Lab Faculty Members Details		
Guide Name: (The Faculty Member Assigned)		Sign: Date:
Grade by Guide:		
Name of Lab Co-Faculty 1		Sign: Date:
Name of Lab Co-Faculty 2		Sign: Date:
Grade by Lab Faculty Members (combined)		
SEE Examiners		
Name of Examiner 1:		Sign: Date:
Name of Examiner 2:		Sign: Date:

Contents

1. Abstract	Pg no
2. Introduction	Pg no
3. Problem Statement.	Pg no
4. Project overview.	Pg no
4.1.Objectives	Pg no
4.2.Goals	Pg no
5. Implementation.	Pg no
4.1.Problem analysis and description.	Pg no
4.2.Modules identified.	Pg no
4.3.Code with comments.	Pg no
6. Output and results	Pg no
7. Conclusions	Pg no
8. References	Pg no

1. Abstract:

An abstract is an outline/brief summary of your paper and your whole project. It should have an intro, body and conclusion. It is a well-developed paragraph, should be exact in wording, and must be understandable to a wide audience. Abstracts highlight major points of your project and explain why your work is important; what your purpose was, how you went about your project, what you learned, and what you concluded.

Keywords: -

(Font size: 12

Fount name: Time new roman in Italic style

Line spacing: 1.15

Abstract should be between 150 to 250 words)

2. Introduction:

The introduction section of a project serves as the opening statement that sets the stage for the entire project. It provides readers with an overview of what to expect, why the project is important, and what motivated its initiation. In other words, the introduction sets the tone for the entire project, providing a roadmap for readers and justifying the project's existence. It should be well-structured, engaging, and informative, motivating readers to continue exploring the project report.

Introduction and Remaining of the document will follow the following format.

Font size: 12

Fount name: Time new roman

Line spacing: 1.15

Introduction should be between approximately ½ to 1 page.

3. Problem statement:

Write your problem statement here.

A problem statement is usually one or two sentences to explain the problem your project will address. In general, a problem statement will outline the negative points of the current situation and explain why these matters. It also serves as a great communication tool, helping to get buy-in and support from others. One of the most important goals of any problem statement is to define the problem being addressed in a way that's clear and precise.

4. Project overview:

Provide the project overview according to the components mentioned here.

A project overview is a detailed description of a project's goals and objectives, the steps to achieve these goals, and the expected outcomes.

- 4.1.Objectives:** An objective describes the desired results of a project, which often includes a tangible item. An objective is specific and measurable, and must meet time, budget, and quality constraints. A project may have one objective, many parallel objectives, or several objectives that must be achieved sequentially. To produce the most benefit, objectives must be defined early in the project life cycle, in phase one, the planning phase.

4.2.Goals: The goal of a project overview is to lay out the details of a project in a concise, easy-to-understand manner that can be presented to clients, team members, and key stakeholders.

5. Project Implementation

Provide the analysis of project, the functions identified to be implemented and finally list the complete commented source code.

Your project group is required to submit a document outlining the project's implementation details. Ensure that your code follows proper coding conventions. Include appropriate comments on critical sections of the code. Overall, your code should have a smooth flow, logical transitions, and should be easy to follow.

5.1. Problem analysis and description.

The problem analysis and description section is a critical component that outlines the specific issue or challenge that the project aims to address. It serves as the foundation upon which the rest of the project is built and helps stakeholders, team members, and anyone else involved or interested in the project to understand the problem and its context.

5.2. Modules identified:

A "module" is a high-level description of a functional area, consisting of a group of processes describing the functionality of the module and a group of packages implementing the functionality.

5.3. Code with comments.

A "code with comments" refers to a piece of computer programming code that includes explanatory comments written alongside the actual code. These comments are meant to provide additional information, explanations, and context about what the code does, how it works, and why certain decisions were made during the coding process. The primary purpose of code comments is to make the code more understandable and maintainable for both the original developer and others who may need to read or modify it in the future.

6. Output and results

Attach the output generated by your project and results. (Screenshot of output and description for results or impact of project)

Outputs" and "results" are two distinct but interconnected aspects of a project, and they are often used to measure the project's success and impact. In short, outputs are the tangible products and deliverables produced during a project's execution, while results are the broader and often longer-term changes or benefits that occur as a direct or indirect consequence of these outputs. Both output and results are essential for evaluating a project's success and effectiveness, with results being the ultimate measure of the project's impact on its intended beneficiaries or stakeholders.

7. Conclusions:

Write the conclusion here.

A conclusion is the last part of something, it means "finally, to sum up," and is used to introduce some final comments at the end of writing.

8. References:

References in a document are citations or sources of information that support and substantiate the content presented in the document. Properly formatted references enhance the credibility and integrity of your writing while giving readers the means to verify and explore the sources you've used.

It's essential to follow the specific citation style guidelines for formatting your references correctly. Additionally, make sure your in-text citations (citations within the main body of your document) correspond to the entries in your references section.

References must be quoted as per the following format:

Author(s) Initial(s). Surname(s), "Title of Report," Abbrev. Name of Co., City of Co., Abbrev. State, Country (abbrev. US State or Country if the City is not 'well known'), Report number/Type (if available), Abbrev. Month. (Day if available), Year of Publication.

Example for reference is given below. Kindly follow the same format for writing reference

G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

K. Elissa, "Title of paper if known," unpublished.

R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetism Japan, p. 301, 1982].

M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

9. Learning Resources:

Reference Books:

1. Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms”, 3rd edition, PHI Learning Private Limited, 2017
2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, “Data Structures and Algorithms”, Pearson.
3. Donald E. Knuth, “The Art of Computer Programming”, Volumes 1 and 3 Pearson.

Web Based Resources and E-books:

1. NPTEL Course on “Design and Analysis of Algorithms”, Prof. Abhiram G Ranade, Prof. Ajit A Diwan and Prof. Sundar Vishwanathan, <https://nptel.ac.in/courses/106101060>
2. “Introduction to Design and Analysis of Algorithms” by Anany Levitin, 2nd edition
<http://160592857366.free.fr/joe/ebooks/ShareData/Anany%20Levitin%20English>
3. https://www.researchgate.net/publication/276847633_A_Review_Report_on_Divide_and_Conquer_Sorting_Algorithm
4. <https://www.ijsrp.org/research-paper-0813/ijsrp-p2014.pdf>
5. <https://iopscience.iop.org/article/10.1088/1742-6596/1566/1/012038/pdf>