1. Basic Analysis of algorithms

a. Implement Sequential Search Algorithm and Analyze its Time Complexity

Sequential Search

Sequential search (also known as linear search) is a simple searching technique used to find an element in an unsorted list. It works by sequentially checking each element in the list until the desired element is found or the list ends.

Algorithm

```
SequentialSearch(A, n, key)
Input: A - array of n elements, key - element to be searched
Output: Index of the key if found, otherwise -1

for i ← 0 to n - 1 do
     if A[i] == key then
          return i  // Return the index of the found element
  return -1  // Key not found in the array
```

Time Complexity Analysis

Let us analyze the worst-case, best-case, and average-case time complexities of the sequential search algorithm.

1. Best Case Analysis (Ω(1))

- In the best case, the key element is found at the first position (`A[0]`).
- The algorithm only requires **one** comparison (`A[0] == key`).
- Hence, the best-case time complexity is **Ω(1)**.

2. Worst Case Analysis (O(n))

- In the worst case, the key element is not present in the array or is the last element

  (`A[n-1]`).
- The algorithm will compare all `n` elements before concluding the search.
- Hence, the worst-case time complexity is **O(n)**.

3. Average Case Analysis (Θ(n))

- In the average case, we assume that the key is equally likely to be found at any position.
- The expected number of comparisons is:

$$C_{\text{avg}} = \frac{1 + 2 + \dots + n}{n} = \frac{n+1}{2}$$

- This simplifies to **Θ(n)**.

Time Complexity

Program:

```
public class BruteForceSearch {
    public static int search(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) return i;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {4, 2, 7, 1, 3};
        System.out.println(search(arr, 7)); // Output: 2
    }
}
OutPut:
```

b. Implement finding Factorial of a given number using recursive me Complexity

Algorithm: Recursive Factorial Computation

The factorial of a non-negative integer n, denoted as **n**!, is defined as:

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

With the base case:

$$0! = 1$$

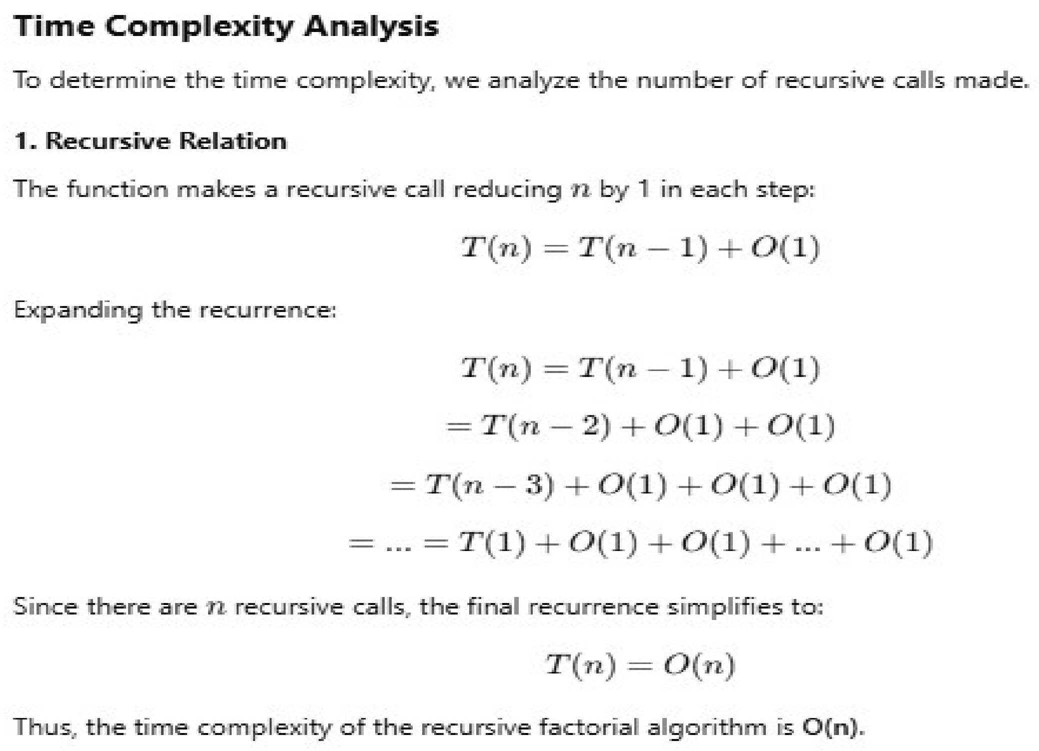
Using recursion, the factorial function can be defined as:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1)!, & \text{if } n > 0 \end{cases}$$

Recursive Algorithm

```
Factorial(n)
Input: A non-negative integer n
Output: Factorial of n
```

```
1. if n == 0 then
2.     return 1
3. else
4.     return n * Factorial(n - 1)
```



Time Complexity Analysis

To determine the time complexity, we analyze the number of recursive calls made.

1. Recursive Relation

The function makes a recursive call reducing n by 1 in each step:

$$T(n) = T(n-1) + O(1)$$

Expanding the recurrence:

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ &= T(n-2) + O(1) + O(1) \\ &= T(n-3) + O(1) + O(1) + O(1) \\ &= \ldots = T(1) + O(1) + O(1) + \ldots + O(1) \end{aligned}$$

Since there are n recursive calls, the final recurrence simplifies to:

$$T(n) = O(n)$$

Thus, the time complexity of the recursive factorial algorithm is **O(n)**.

Complexity

Best, Worst, and Average Case Analysis

Case	Time Complexity	Explanation
Best Case	**O(n)**	The algorithm always runs exactly `n` recursive calls, no matter what value is passed.
Worst Case	**O(n)**	The function recursively calls itself `n` times before reaching the base case.
Average Case	**O(n)**	The number of recursive calls remains the same in all cases.

Program:

```
public class FactorialRecursive {
    // Recursive function to calculate factorial
    public static long factorial(int n) {
        if (n == 0) { // Base case
            return 1;
```

```java
        }
        return n * factorial(n - 1); // Recursive call
    }
    // Main method to test the function
    public static void main(String[] args) {
        int num = 5; // Example input
        System.out.println("Factorial of " + num + " is: " + factorial(num));
    }
}
```

2. Brute force Technique

2a. Implement selection sort algorithm and Analyze its Time Complexity

Algorithm: Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It divides the array into two parts:

The algorithm iteratively selects the smallest element from the unsorted part and swaps it with the first element of the unsorted part, growing the sorted part by one element.

Pseudocode

```
SelectionSort(A, n)
Input: A - array of n elements
Output: Sorted array A

 for i ← 0 to n - 2 do
     minIndex ← i
     for j ← i + 1 to n - 1 do
         if A[j] < A[minIndex] then
             minIndex ← j
     Swap A[i] and A[minIndex]
```

Time Complexity Analysis

1. Best Case ($\Omega(n^2)$)

- The selection sort algorithm performs **n-1 iterations** of the outer loop.
- The inner loop runs **n-1, n-2, ..., 1** times for each iteration.
- Comparisons are independent of the array's initial order, so the number of comparisons remains the same regardless of input.

$$T(n) = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} \approx O(n^2)$$

2. Worst Case ($O(n^2)$)

- The worst case occurs when the array is in descending order, but selection sort's performance is not affected by the input order.
- The number of comparisons is still the same as in the best case:

$$T(n) = \frac{n(n-1)}{2} \approx O(n^2)$$

3. Average Case (Θ(n^2))

- The average case assumes a random order of elements in the array.
- Since the structure of selection sort does not change based on input, the number of comparisons remains the same:

$$T(n) = \frac{n(n-1)}{2} \approx O(n^2)$$

Time Complexities:

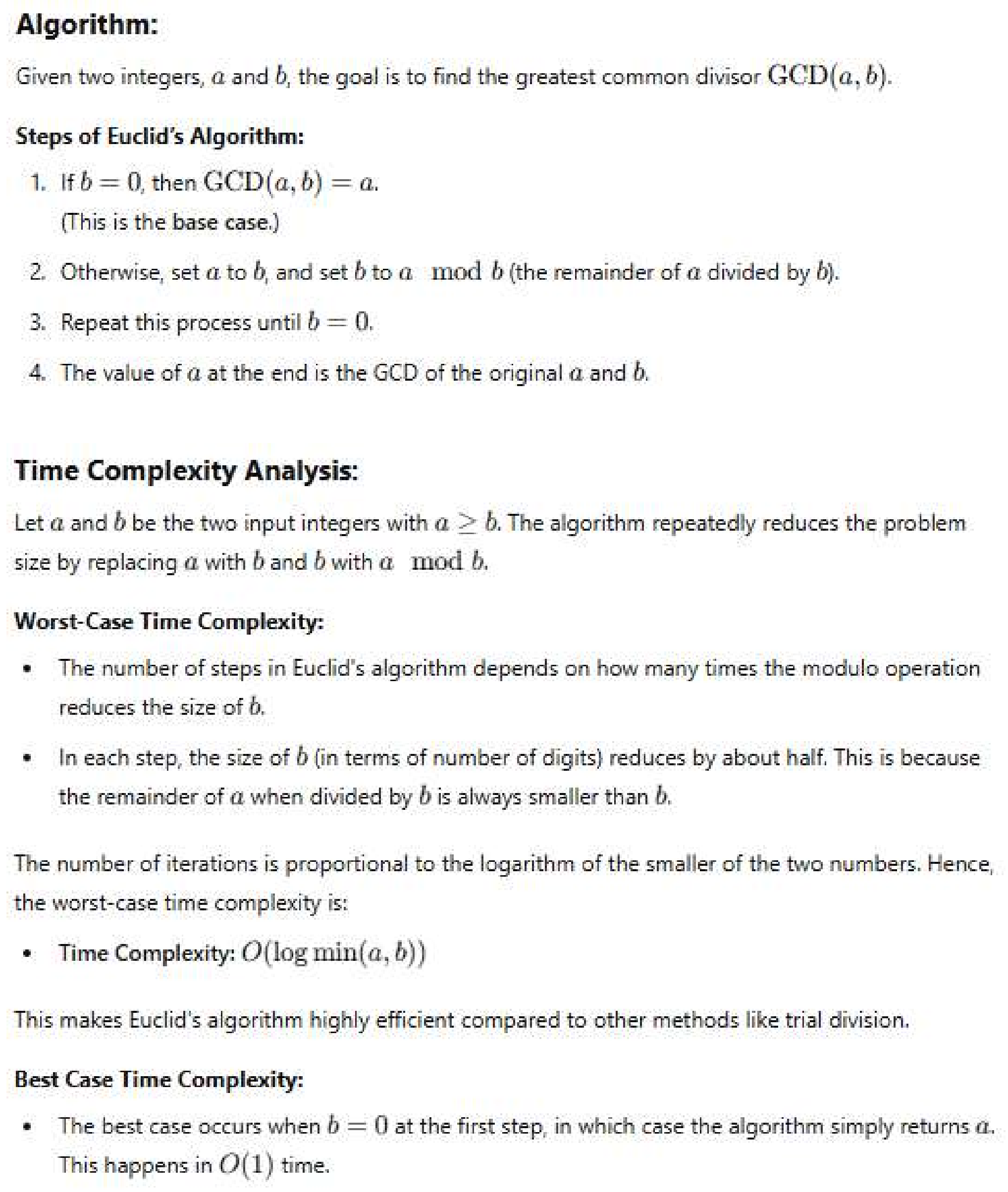
Program:

```
public class SelectionSort {

    // Method to perform Selection Sort

    public static void selectionSort(int[] arr) {

        int n = arr.length;


        // Traverse through all elements in the array

        for (int i = 0; i < n - 1; i++) {

            // Find the minimum element in unsorted array

            int minIndex = i;


            for (int j = i + 1; j < n; j++) {

                if (arr[j] < arr[minIndex]) {

                    minIndex = j; // Update minIndex if a
smaller element is found

                }

            }


            // Swap the found minimum element with the first
element
```

```java
            swap(arr, i, minIndex);
        }
    }
    // Helper method to swap two elements
    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
    // Method to print the array
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
    // Main method to test the Selection Sort
    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};

        System.out.println("Unsorted Array:");
        printArray(arr);
        selectionSort(arr); // Perform Selection Sort
        System.out.println("Sorted Array:");
        printArray(arr);
        }

}
```

2b. Implement Euclid's algorithm and Analyze its Time Complexity

Euclid's algorithm is an efficient method for finding the Greatest Common Divisor (GCD) of two integers. The algorithm repeatedly replaces the larger number with its remainder when divided by the smaller number, until the remainder becomes zero. The divisor at that point is the GCD.



Algorithm:

Given two integers, a and b, the goal is to find the greatest common divisor $\mathrm{GCD}(a, b)$.

Steps of Euclid's Algorithm:

1. If $b = 0$, then $\mathrm{GCD}(a, b) = a$.
   (This is the base case.)
2. Otherwise, set a to b, and set b to $a \mod b$ (the remainder of a divided by b).
3. Repeat this process until $b = 0$.
4. The value of a at the end is the GCD of the original a and b.

Time Complexity Analysis:

Let a and b be the two input integers with $a \geq b$. The algorithm repeatedly reduces the problem size by replacing a with b and b with $a \mod b$.

Worst-Case Time Complexity:

- The number of steps in Euclid's algorithm depends on how many times the modulo operation reduces the size of b.
- In each step, the size of b (in terms of number of digits) reduces by about half. This is because the remainder of a when divided by b is always smaller than b.

The number of iterations is proportional to the logarithm of the smaller of the two numbers. Hence, the worst-case time complexity is:

- **Time Complexity:** $O(\log \min(a, b))$

This makes Euclid's algorithm highly efficient compared to other methods like trial division.

Best Case Time Complexity:

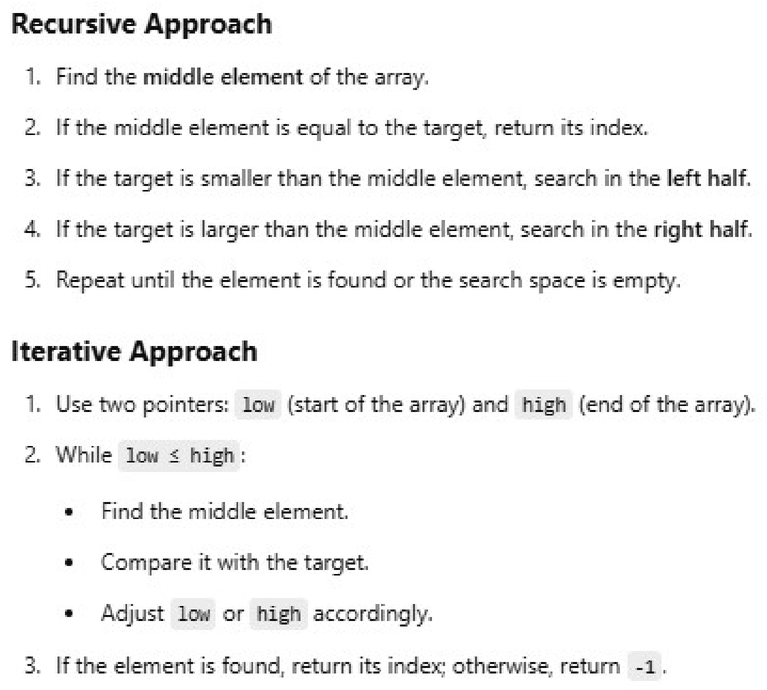
- The best case occurs when $b = 0$ at the first step, in which case the algorithm simply returns a. This happens in $O(1)$ time.

Program:

```
public class EuclidGCD {

    // Method to find the GCD of two numbers using Euclid's
Algorithm

    public static int euclidGCD(int a, int b) {

        // Base case: if b is 0, then GCD is a

        if (b == 0) {

            return a;

        }

        // Recursively apply the Euclid's Algorithm

        return euclidGCD(b, a % b);

    }

    public static void main(String[] args) {

        // Example input

        int a = 56;

        int b = 98;

        // Output the GCD of a and b

        int gcd = euclidGCD(a, b);

        System.out.println("The GCD of " + a + " and " + b + "
is: " + gcd);

    }

}
```

3.Decrease-and-Conquer Method

3a. Implement Binary search algorithm and Analyze its Time Complexity



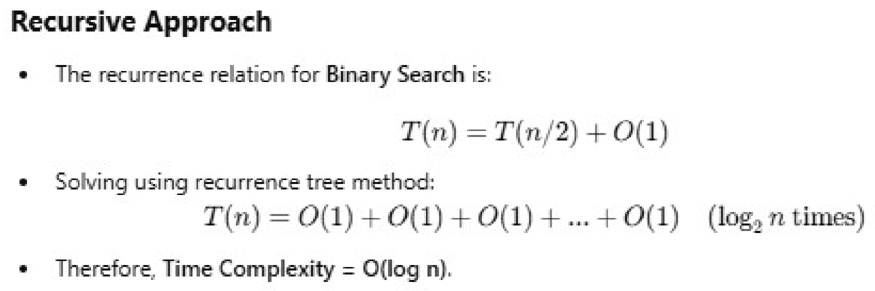
Recursive Approach

1. Find the **middle element** of the array.
2. If the middle element is equal to the target, return its index.
3. If the target is smaller than the middle element, search in the **left half**.
4. If the target is larger than the middle element, search in the **right half**.
5. Repeat until the element is found or the search space is empty.

Iterative Approach

1. Use two pointers: `low` (start of the array) and `high` (end of the array).
2. While `low ≤ high`:
   - Find the middle element.
   - Compare it with the target.
   - Adjust `low` or `high` accordingly.
3. If the element is found, return its index; otherwise, return `-1`.

Time Complexity Analysis



Recursive Approach

- The recurrence relation for **Binary Search** is:

$$T(n) = T(n/2) + O(1)$$

- Solving using recurrence tree method:

$$T(n) = O(1) + O(1) + O(1) + ... + O(1) \quad (\log_2 n \text{ times})$$

- Therefore, **Time Complexity = O(log n)**.

```
public class BinarySearch {

    public static int search(int[] arr, int target, int low,
int high) {

        if (low > high) return -1;

        int mid = (low + high) / 2;

        if (arr[mid] == target) return mid;

        else if (arr[mid] < target) return search(arr, target,
mid + 1, high);

        else return search(arr, target, low, mid - 1);

    }
    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9};
        System.out.println(search(arr, 5, 0, arr.length - 1));
// Output: 2
    }
}
```

4. Divide-and-Conquer Technique

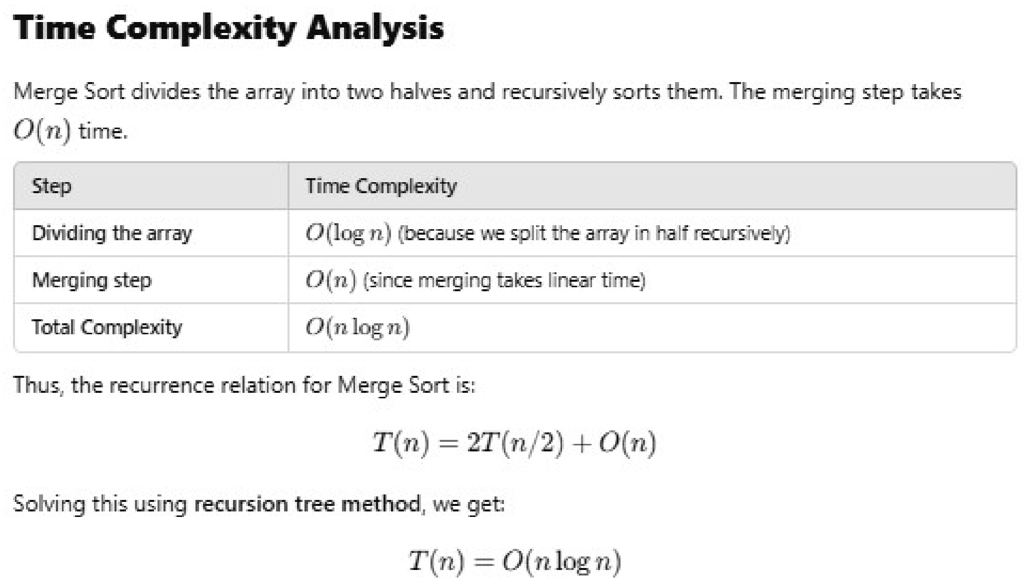
4a. Implement Merge sort algorithm and Analyze its Time Complexity

Merge Sort is a **divide and conquer** algorithm that recursively splits an array into two halves, sorts each half, and then merges the sorted halves back together. It is an efficient, stable sorting algorithm that guarantees a worst-case time complexity of O(nlogn).

Algorithm:

```
MERGE_SORT(A, left, right)
1. If left < right:
2.     mid = (left + right) / 2
3.     MERGE_SORT(A, left, mid)   // Recursively sort left half
4.     MERGE_SORT(A, mid+1, right)  // Recursively sort right half
5.     MERGE(A, left, mid, right)  // Merge the sorted halves


MERGE(A, left, mid, right)
1. Create leftSubArray = A[left...mid]
2. Create rightSubArray = A[mid+1...right]
3. i = 0, j = 0, k = left   // Initial indices for left, right, and merged array
4. While i < size(leftSubArray) and j < size(rightSubArray):
5.     If leftSubArray[i] ≤ rightSubArray[j]:
6.         A[k] = leftSubArray[i]
7.         i = i + 1
8.     Else:
9.         A[k] = rightSubArray[j]
```

```
10.       j = j + 1
11.    k = k + 1
12. Copy remaining elements of leftSubArray (if any) to A
13. Copy remaining elements of rightSubArray (if any) to A
```



Time Complexity Analysis

Merge Sort divides the array into two halves and recursively sorts them. The merging step takes $O(n)$ time.

Step	Time Complexity
Dividing the array	$O(\log n)$ (because we split the array in half recursively)
Merging step	$O(n)$ (since merging takes linear time)
Total Complexity	$O(n \log n)$

Thus, the recurrence relation for Merge Sort is:

$$T(n) = 2T(n/2) + O(n)$$

Solving this using **recursion tree method**, we get:

$$T(n) = O(n \log n)$$

Program:

```
import java.util.Arrays;


public class MergeSort {
    public static void mergeSort(int[] arr) {
        if (arr.length <= 1) return;
        int mid = arr.length / 2;
        int[] left = Arrays.copyOfRange(arr, 0, mid);
        int[] right = Arrays.copyOfRange(arr, mid, 
arr.length);


        mergeSort(left);
        mergeSort(right);
        merge(arr, left, right);
    }
    private static void merge(int[] arr, int[] left, int[]
right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] < right[j]) arr[k++] = left[i++];
            else arr[k++] = right[j++];
        }
        while (i < left.length) arr[k++] = left[i++];
        while (j < right.length) arr[k++] = right[j++];
    }

    public static void main(String[] args) {
        int[] arr = {6, 3, 8, 5, 2};
        mergeSort(arr);
        System.out.println(Arrays.toString(arr)); // Output:
[2, 3, 5, 6, 8]
    }
}
```

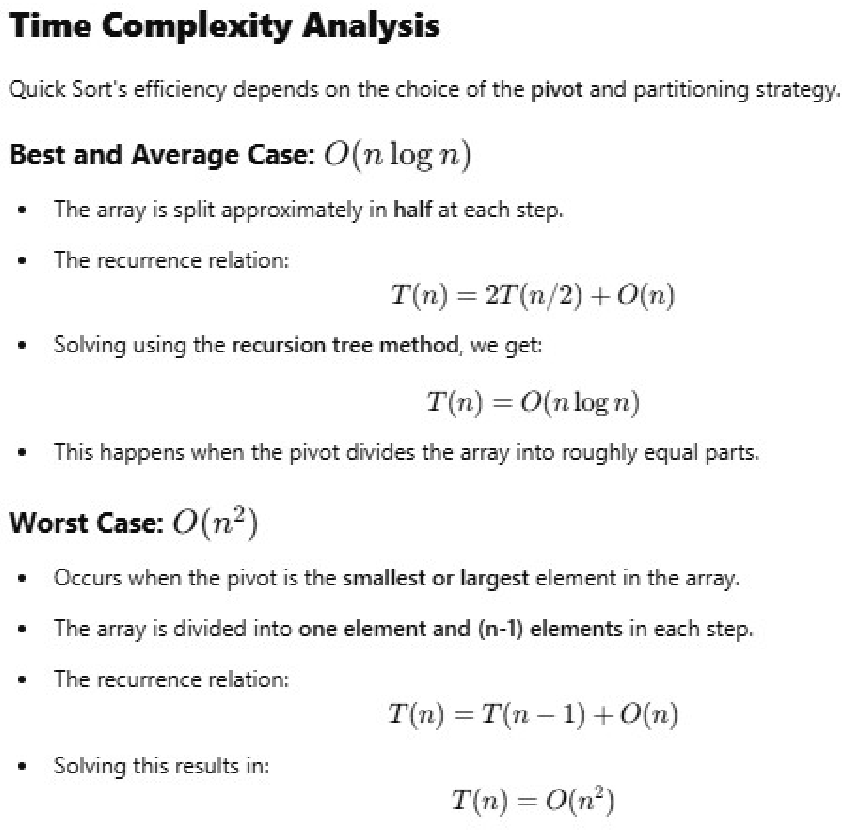
4b. Implement Quick sort algorithm and Analyze its Time Complexity

Quick Sort is an efficient **divide and conquer** sorting algorithm that selects a **pivot** element, partitions the array around the pivot, and recursively sorts the partitions. It is **in-place** and has an average time complexity of ***O(nlogn)***.

Algorithm:

```
QuickSort(A, low, high)
If low < high then:
p ← Partition(A, low, high)   // Find pivot position
QuickSort(A, low, p - 1)  // Recursively sort left partition
QuickSort(A, p + 1, high) // Recursively sort right partition

Partition(A, low, high)
Choose pivot (e.g., A[high])
Set i ← low - 1
```

```
For j ← low to high - 1 do:
    If A[j] ≤ pivot then:
        i ← i + 1
        Swap A[i] and A[j]
Swap A[i + 1] and A[high]
Return i + 1  // Pivot index
```



Time Complexity Analysis

Quick Sort's efficiency depends on the choice of the **pivot** and partitioning strategy.

Best and Average Case: $O(n \log n)$

- The array is split approximately in **half** at each step.
- The recurrence relation:
$$T(n) = 2T(n/2) + O(n)$$
- Solving using the **recursion tree method**, we get:
$$T(n) = O(n \log n)$$
- This happens when the pivot divides the array into roughly equal parts.

Worst Case: $O(n^2)$

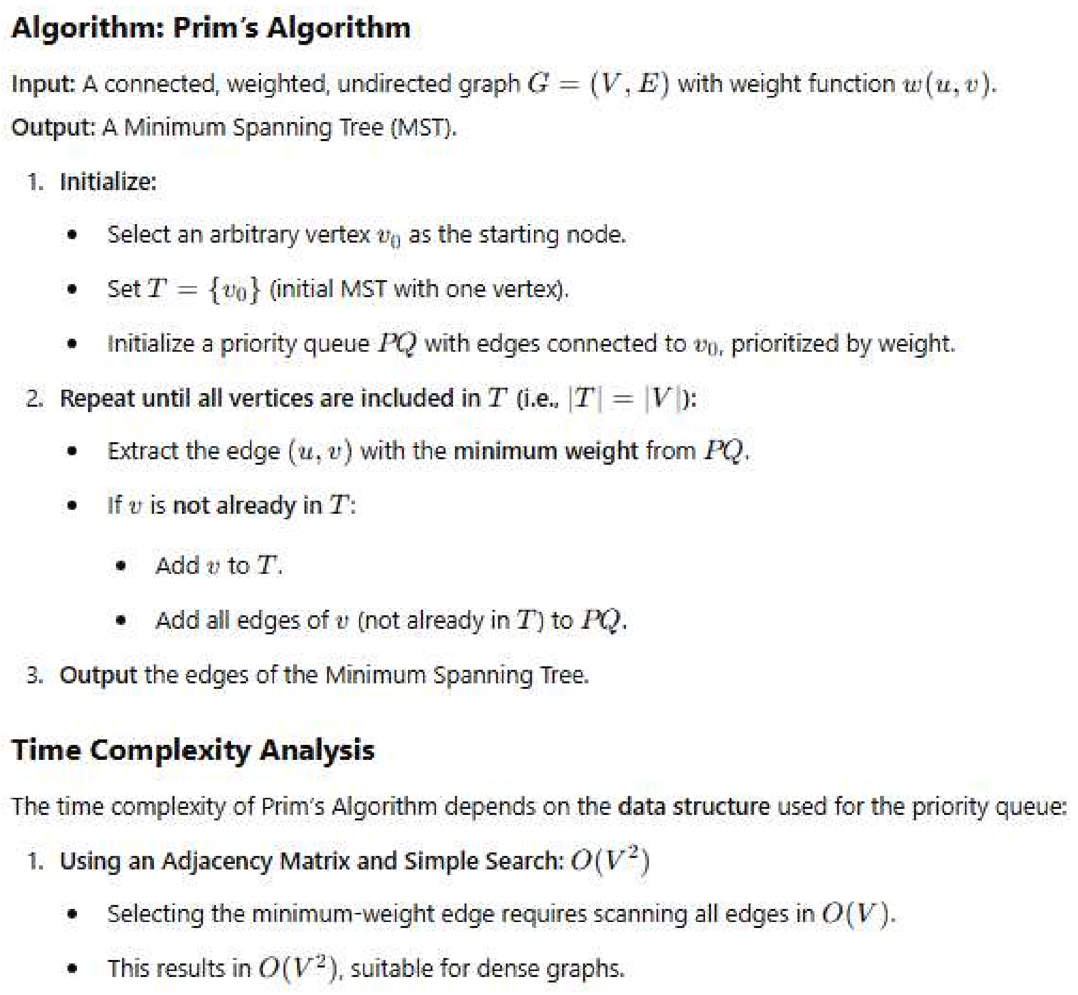
- Occurs when the pivot is the **smallest or largest** element in the array.
- The array is divided into **one element and (n-1) elements** in each step.
- The recurrence relation:
$$T(n) = T(n-1) + O(n)$$
- Solving this results in:
$$T(n) = O(n^2)$$

Program:

```
public class QuickSort {


    // Method to perform QuickSort
    public static void quickSort(int[] arr, int low, int high)
{
        if (low < high) {
            // Find the partition index
            int pivotIndex = partition(arr, low, high);
            // Recursively sort elements before and after
partition
            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    // Partition function
    public static int partition(int[] arr, int low, int high)
{
        int pivot = arr[high];  // Choosing the last element
as pivot
        int i = low - 1; // Index of smaller element

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) { // If current element is
smaller than pivot
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high); // Swap pivot to correct
position
        return i + 1; // Return pivot index
    }

    // Swap function
    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
    // Main method to test Quick Sort
    public static void main(String[] args) {
        int[] arr = { 10, 7, 8, 9, 1, 5 };
        int n = arr.length;
        System.out.println("Unsorted Array:");
        printArray(arr);
        quickSort(arr, 0, n - 1);
        System.out.println("Sorted Array:");
        printArray(arr);
    }

    // Method to print array
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

5.Greedy Method

5a. Implement prim’s algorithm and Analyze its Time Complexity



Algorithm: Prim's Algorithm

Input: A connected, weighted, undirected graph $G = (V, E)$ with weight function $w(u, v)$.

Output: A Minimum Spanning Tree (MST).

1. **Initialize:**
   - Select an arbitrary vertex v_0 as the starting node.
   - Set $T = \{v_0\}$ (initial MST with one vertex).
   - Initialize a priority queue PQ with edges connected to v_0, prioritized by weight.
2. **Repeat until all vertices are included in T (i.e., $|T| = |V|$):**
   - Extract the edge (u, v) with the **minimum weight** from PQ.
   - If v is **not already** in T:
     - Add v to T.
     - Add all edges of v (not already in T) to PQ.
3. **Output** the edges of the Minimum Spanning Tree.

Time Complexity Analysis

The time complexity of Prim's Algorithm depends on the **data structure** used for the priority queue:

1. **Using an Adjacency Matrix and Simple Search: $O(V^2)$**
   - Selecting the minimum-weight edge requires scanning all edges in $O(V)$.
   - This results in $O(V^2)$, suitable for dense graphs.

Program:

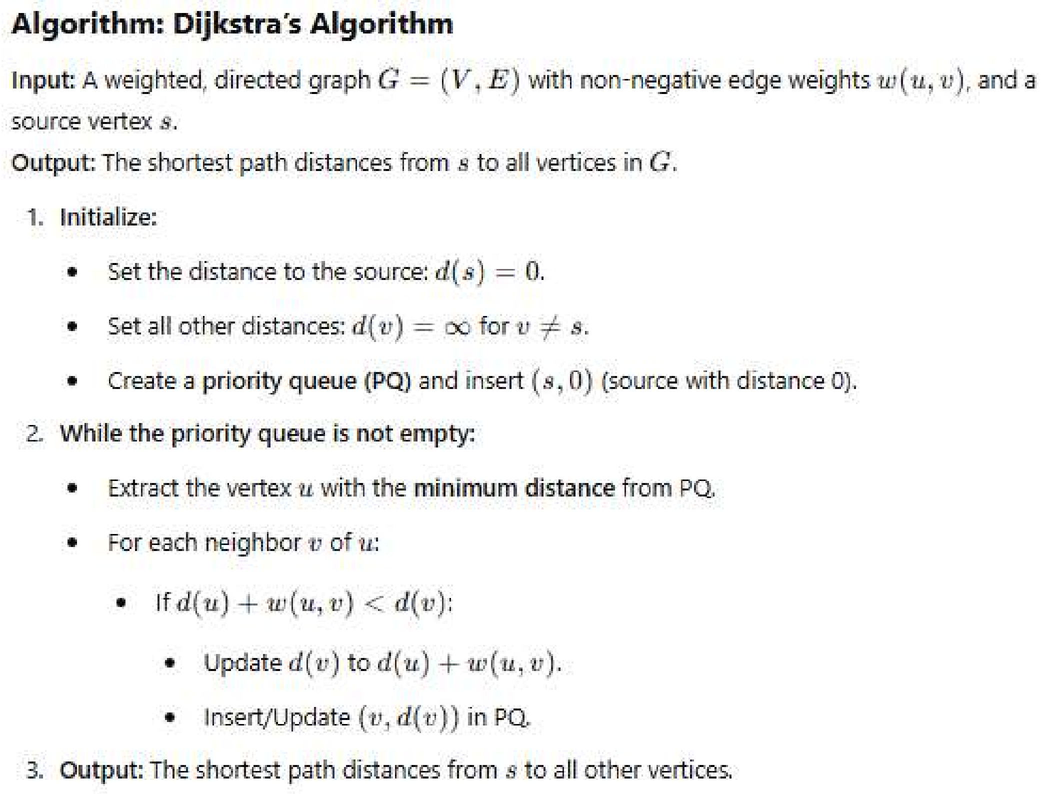
```
import java.util.Scanner;
public class PrimsAlgorithm1 {

    static final int INF = 9999;
    static final int MAX = 20;
    static int[][] G = new int[MAX][MAX];
    static int[][] spanning = new int[MAX][MAX];
    static int n;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of vertices: ");
        n = scanner.nextInt();

        System.out.println("\nEnter the adjacency matrix:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                G[i][j] = scanner.nextInt();
            }
        }

        int totalCost = prims();

        System.out.println("\nSpanning tree matrix:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(spanning[i][j] + "\t");
            }
            System.out.println();
        }

        System.out.println("\nTotal cost of the spanning tree = " +
totalCost);
    }

    static int prims() {
        int[][] cost = new int[MAX][MAX];
        int[] distance = new int[MAX];
        int[] from = new int[MAX];
        int[] visited = new int[MAX];
        int minCost = 0;

        // Create the cost matrix and initialize the spanning matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (G[i][j] == 0) {
                    cost[i][j] = INF;
                } else {
                    cost[i][j] = G[i][j];
                }
                spanning[i][j] = 0;
            }
        }

        // Initialize distance, visited, and from arrays
        distance[0] = 0;
        visited[0] = 1;
        for (int i = 1; i < n; i++) {
            distance[i] = cost[0][i];
            from[i] = 0;
            visited[i] = 0;
        }

        int noOfEdges = n - 1;
        while (noOfEdges > 0) {
            int minDistance = INF, v = -1;

            // Find the vertex at the minimum distance from the tree
            for (int i = 1; i < n; i++) {
                if (visited[i] == 0 && distance[i] < minDistance) {
                    v = i;
                    minDistance = distance[i];
                }
            }

            int u = from[v];

            // Add the edge to the spanning tree
            spanning[u][v] = distance[v];
            spanning[v][u] = distance[v];
            noOfEdges--;
            visited[v] = 1;

            // Update the distance array
            for (int i = 1; i < n; i++) {
                if (visited[i] == 0 && cost[i][v] < distance[i]) {
                    distance[i] = cost[i][v];
                    from[i] = v;
                }
            }

            minCost += cost[u][v];
        }

        return minCost;
    }
}

/* OUTPUT
 * PS D:\MTech_ADA_LabPgms\ADA_LAB> javac PrimsAlgorithm1.java
PS D:\MTech_ADA_LabPgms\ADA_LAB> java  PrimsAlgorithm1
Enter the number of vertices: 3

Enter the adjacency matrix:
0 1 1
1 0 1
1 1 0

Spanning tree matrix:
0       1       1
1       0       0
1       0       0
```

```
Total cost of the spanning tree = 2

 */
```

5b. Dijkstra’s algorithm and Analyze its Time Complexity

Dijkstra’s Algorithm

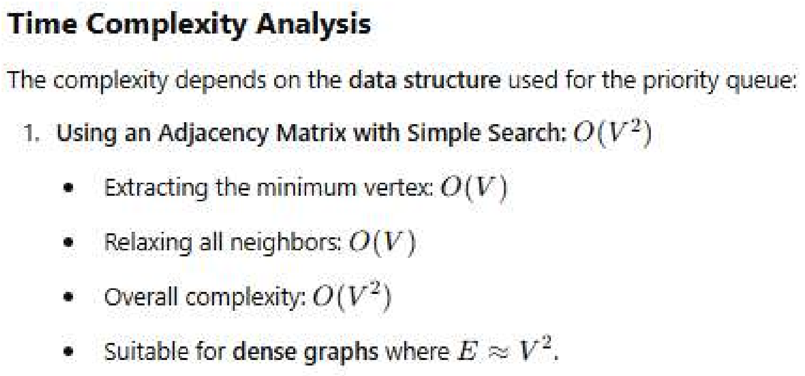
Dijkstra’s algorithm is a greedy algorithm used to find the shortest path from a single **source** to all other vertices in a weighted graph with non-negative edge weights.



Algorithm: Dijkstra′s Algorithm

Input: A weighted, directed graph $G = (V, E)$ with non-negative edge weights $w(u, v)$, and a source vertex s.
Output: The shortest path distances from s to all vertices in G.

1. **Initialize:**
   - Set the distance to the source: $d(s) = 0$.
   - Set all other distances: $d(v) = \infty$ for $v \neq s$.
   - Create a **priority queue (PQ)** and insert $(s, 0)$ (source with distance 0).
2. **While the priority queue is not empty:**
   - Extract the vertex u with the **minimum distance** from PQ.
   - For each neighbor v of u:
     - If $d(u) + w(u, v) < d(v)$:
       - Update $d(v)$ to $d(u) + w(u, v)$.
       - Insert/Update $(v, d(v))$ in PQ.
3. **Output:** The shortest path distances from s to all other vertices.



Time Complexity Analysis

The complexity depends on the **data structure** used for the priority queue:

1. **Using an Adjacency Matrix with Simple Search:** $O(V^2)$
   - Extracting the minimum vertex: $O(V)$
   - Relaxing all neighbors: $O(V)$
   - Overall complexity: $O(V^2)$
   - Suitable for **dense graphs** where $E \approx V^2$.

Program:

```
 public class DijkstraAlgorithm {

          public void dijkstraAlgorithm(int[][] graph, int source) {
          // number of nodes
          int nodes = graph.length;
          boolean[] visited_vertex = new boolean[nodes];
          int[] dist = new int[nodes];
          for (int i = 0; i < nodes; i++) {
            visited_vertex[i] = false;
            dist[i] = Integer.MAX_VALUE;
          }

          // Distance of self loop is zero
          dist[source] = 0;
          for (int i = 0; i < nodes; i++) {

            // Updating the distance between neighboring vertex and source
vertex
            int u = find_min_distance(dist, visited_vertex);
            visited_vertex[u] = true;

            // Updating the distances of all the neighboring vertices
            for (int v = 0; v < nodes; v++) {
              if (!visited_vertex[v] && graph[u][v] != 0 && (dist[u] +
graph[u][v] < dist[v])) {
                dist[v] = dist[u] + graph[u][v];
              }
            }
          }
          for (int i = 0; i < dist.length; i++) {
            System.out.println(String.format("Distance from Vertex %s to
Vertex %s is %s", source, i, dist[i]));
          }

        }

        // defining the method to find the minimum distance
        private static int find_min_distance(int[] dist, boolean[]
visited_vertex) {
          int minimum_distance = Integer.MAX_VALUE;
          int minimum_distance_vertex = -1;
          for (int i = 0; i < dist.length; i++) {
            if (!visited_vertex[i] && dist[i] < minimum_distance) {
              minimum_distance = dist[i];
              minimum_distance_vertex = i;
            }
          }
          return minimum_distance_vertex;
        }

          public static void main(String[] args) {
          // declaring the nodes of the graphs
          int graph[][] = new int[][] {
            { 0, 1, 1, 2, 0, 0, 0 },
            { 0, 0, 2, 0, 0, 3, 0 },
            { 1, 2, 0, 1, 3, 0, 0 },
            { 2, 0, 1, 0, 2, 0, 1 },
            { 0, 0, 3, 0, 0, 2, 0 },
            { 0, 3, 0, 0, 2, 0, 1 },
            { 0, 2, 0, 1, 0, 1, 0 }
          };

          DijkstraAlgorithm Test = new DijkstraAlgorithm();

          Test.dijkstraAlgorithm(graph, 0);
        }
      }

      /*OUTPUT:
      Distance from Vertex 0 to Vertex 0 is 0
Distance from Vertex 0 to Vertex 1 is 1
Distance from Vertex 0 to Vertex 2 is 1
Distance from Vertex 0 to Vertex 3 is 2
Distance from Vertex 0 to Vertex 4 is 4
Distance from Vertex 0 to Vertex 5 is 4
Distance from Vertex 0 to Vertex 6 is 3

       */
```

Pgm6 a &b : Implement Warshall and Floyd's algorithm

Warshall's and Floyd's Algorithm

Warshall's Algorithm is used to compute the **transitive closure** of a directed graph, whereas **Floyd's Algorithm (Floyd-Warshall Algorithm)** is used to find the **shortest paths between all pairs of vertices** in a weighted graph.

1. Warshall´s Algorithm (Transitive Closure Algorithm)

Warshall's Algorithm determines the **reachability matrix** of a directed graph, which tells whether there is a path between two vertices.

Algorithm: Warshall's Algorithm

Input: A directed graph represented as an adjacency matrix $R(0)$, where:

- $R(0)[i][j] = 1$ if there is a direct edge from i to j, else 0.

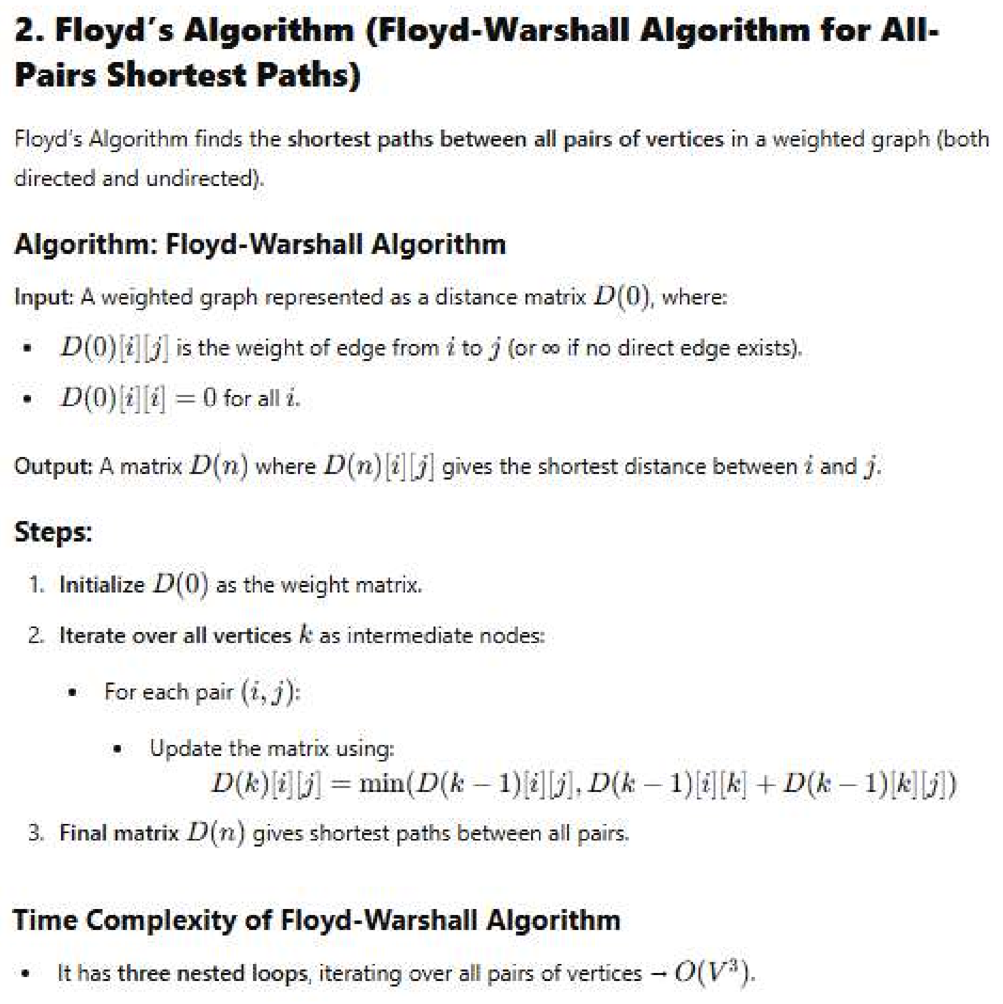
Output: The transitive closure matrix $R(n)$, where $R(n)[i][j] = 1$ if there exists a path from i to j, otherwise 0.

Steps:

1. **Initialize** $R(0)$ as the adjacency matrix.
2. **Iterate over all vertices** k as intermediate nodes:
    - For each pair of vertices (i, j):
        - Update the matrix:
          $$R(k)[i][j] = R(k-1)[i][j] \; OR \; (R(k-1)[i][k] \; AND \; R(k-1)[k][j])$$
3. **Final matrix** $R(n)$ gives the transitive clo of the graph.

Time Complexity of Warshall's Algorithm

- The algorithm runs three nested loops over V vertices → $O(V^3)$.



2. Floyd's Algorithm (Floyd-Warshall Algorithm for All-Pairs Shortest Paths)

Floyd's Algorithm finds the **shortest paths between all pairs of vertices** in a weighted graph (both directed and undirected).

Algorithm: Floyd-Warshall Algorithm

Input: A weighted graph represented as a distance matrix $D(0)$, where:

- $D(0)[i][j]$ is the weight of edge from i to j (or ∞ if no direct edge exists).
- $D(0)[i][i] = 0$ for all i.

Output: A matrix $D(n)$ where $D(n)[i][j]$ gives the shortest distance between i and j.

Steps:

1. **Initialize** $D(0)$ as the weight matrix.
2. **Iterate over all vertices** k as intermediate nodes:
   - For each pair (i, j):
     - Update the matrix using:
       $$D(k)[i][j] = \min(D(k-1)[i][j], D(k-1)[i][k] + D(k-1)[k][j])$$
3. **Final matrix** $D(n)$ gives shortest paths between all pairs.

Time Complexity of Floyd-Warshall Algorithm

- It has **three nested loops**, iterating over all pairs of vertices → $O(V^3)$.

Program:

```
import java.lang.*;

public class AllPairShortestPath {
    final static int INF = 99999, V = 4;

    void floydWarshall(int dist[][])
    {

        int i, j, k;

        /* Add all vertices one by one  to the set of intermediate vertices.
          ---> Before start of an iteration,
               we have shortest
               distances between all pairs
               of vertices such that
               the shortest distances consider
               only the vertices in
```

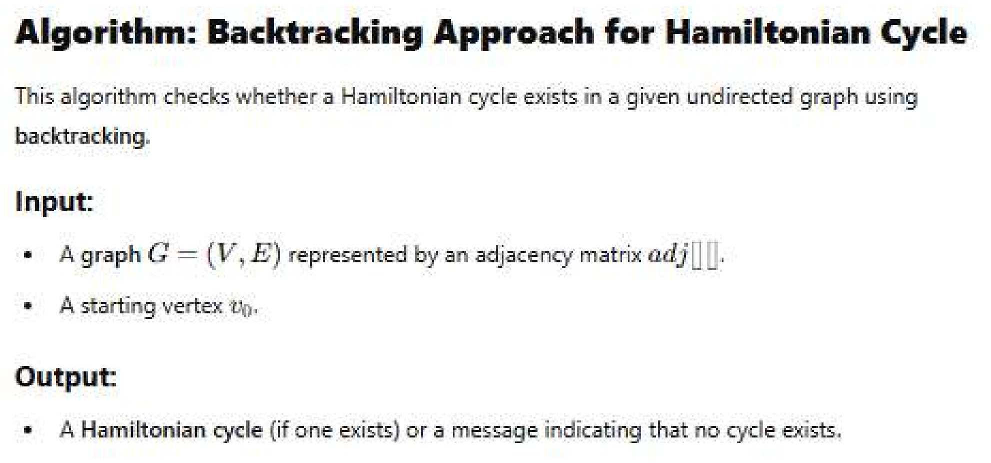
```
              set {0, 1, 2, .. k-1} as
              intermediate vertices.
         ----> After the end of an iteration,
               vertex no. k is added
               to the set of intermediate
               vertices and the set
               becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++) {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++) {
                // Pick all vertices as destination for the
                // above picked source
                for (j = 0; j < V; j++) {
                    // If vertex k is on the shortest path
                    // from i to j, then update the value of
                    // dist[i][j]
                    if (dist[i][k] + dist[k][j]
                        < dist[i][j])
                        dist[i][j]
                            = dist[i][k] + dist[k][j];
                }
            }
        }

        // Print the shortest distance matrix
        printSolution(dist);
    }

    void printSolution(int dist[][])
    {
        System.out.println(
            "The following matrix shows the shortest "
            + "distances between every pair of vertices");
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
                if (dist[i][j] == INF)
                    System.out.print("INF ");
                else
                    System.out.print(dist[i][j] + "   ");
            }
            System.out.println();
        }
    }

    // Driver's code
    public static void main(String[] args)
    {
        /* Let us create the following weighted graph
```

```
                10
            (0)------->(3)
            |         /|\
            5 |          |
            |          | 1
            \|/         |
            (1)------->(2)
                3           */
            int graph[][] = { { 0, 5, INF, 10 },
                              { INF, 0, 3, INF },
                              { INF, INF, 0, 1 },
                              { INF, INF, INF, 0 } };
            AllPairShortestPath a = new AllPairShortestPath();

            // Function call
            a.floydWarshall(graph);
        }
    }

/* ----OUTPUT
The following matrix shows the shortest distances between every pair of
vertices
0   5   8   9
INF 0   3   4
INF INF 0   1
INF INF INF 0  */
```

Backtracking

7a. Implement Hamiltonian cycles algorithm

A Hamiltonian cycle in a graph is a cycle that visits every vertex exactly once and returns to the starting vertex. The problem of finding such a cycle is NP-complete, meaning that no known polynomial-time algorithm exists for solving it in general cases.



Algorithm: Backtracking Approach for Hamiltonian Cycle

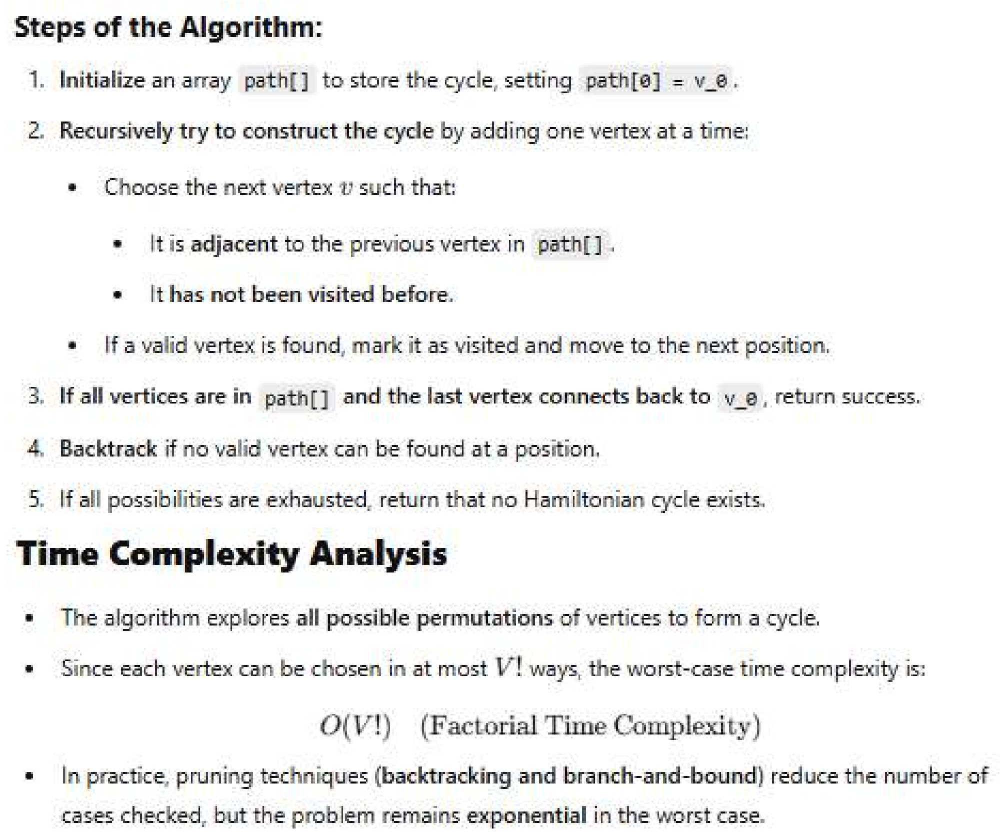
This algorithm checks whether a Hamiltonian cycle exists in a given undirected graph using backtracking.

Input:

- A graph $G = (V, E)$ represented by an adjacency matrix $adj[][]$.
- A starting vertex v_0.

Output:

- A **Hamiltonian cycle** (if one exists) or a message indicating that no cycle exists.



Steps of the Algorithm:

1. **Initialize** an array `path[]` to store the cycle, setting `path[0] = v_0`.
2. **Recursively try to construct the cycle** by adding one vertex at a time:
   - Choose the next vertex v such that:
     - It is **adjacent** to the previous vertex in `path[]`.
     - **It has not been visited before.**
   - If a valid vertex is found, mark it as visited and move to the next position.
3. **If all vertices are in `path[]` and the last vertex connects back to `v_0`**, return success.
4. **Backtrack** if no valid vertex can be found at a position.
5. If all possibilities are exhausted, return that no Hamiltonian cycle exists.

Time Complexity Analysis

- The algorithm explores **all possible permutations** of vertices to form a cycle.
- Since each vertex can be chosen in at most $V!$ ways, the worst-case time complexity is:

$$O(V!) \quad \text{(Factorial Time Complexity)}$$

- In practice, pruning techniques (**backtracking and branch-and-bound**) reduce the number of cases checked, but the problem remains **exponential** in the worst case.

Program:

```
public class HamiltonianCycle {

    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
       added at index 'pos'in the Hamiltonian Cycle
       constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][], int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
           the previously added vertex. */
        if (graph[path[pos - 1]][v] == 0)
            return false;

        /* Check if the vertex has already been included.
           This step can be optimized by creating an array
           of size V */
        for (int i = 0; i < pos; i++)
            if (path[i] == v)
                return false;

        return true;
    }

    /* A recursive utility function to solve hamiltonian
       cycle problem */
    boolean hamCycleUtil(int graph[][], int path[], int pos)
    {
        /* base case: If all vertices are included in
           Hamiltonian Cycle */
        if (pos == V)
        {
            // And if there is an edge from the last included
            // vertex to the first vertex
            if (graph[path[pos - 1]][path[0]] == 1)
                return true;
            else
                return false;
        }

        // Try different vertices as a next candidate in
        // Hamiltonian Cycle. We don't try for 0 as we
        // included 0 as starting point in hamCycle()
        for (int v = 1; v < V; v++)
        {
            /* Check if this vertex can be added to Hamiltonian
               Cycle */
            if (isSafe(v, graph, path, pos))
            {
                path[pos] = v;

                /* recur to construct rest of the path */
                if (hamCycleUtil(graph, path, pos + 1) == true)
                    return true;

                /* If adding vertex v doesn't lead to a solution,
                   then remove it */
                path[pos] = -1;
            }
        }

        /* If no vertex can be added to Hamiltonian Cycle
           constructed so far, then return false */
        return false;
    }

    /* This function solves the Hamiltonian Cycle problem using
       Backtracking. It mainly uses hamCycleUtil() to solve the
       problem. It returns false if there is no Hamiltonian Cycle
       possible, otherwise return true and prints the path.
```

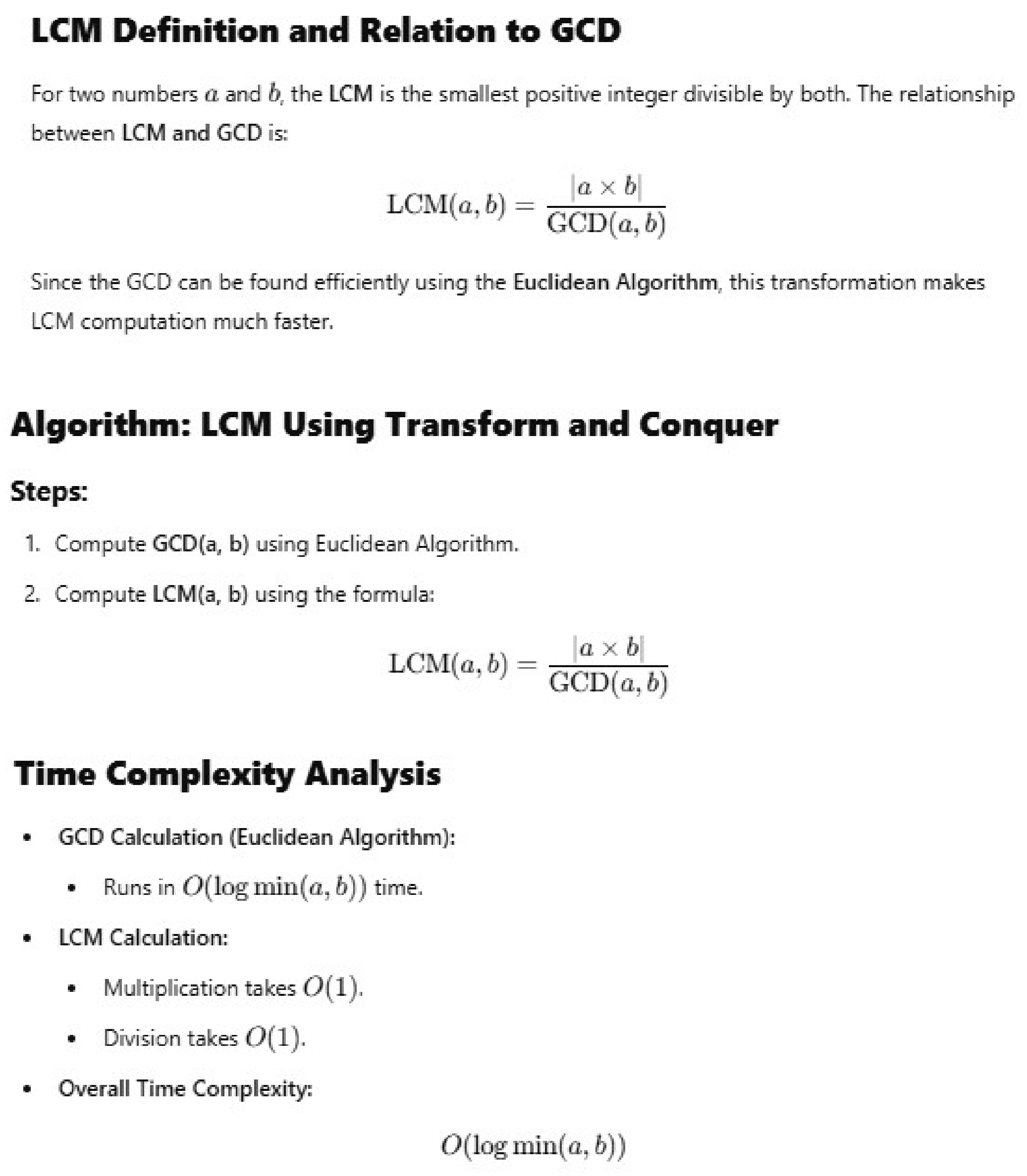
```
       Please note that there may be more than one solutions,
       this function prints one of the feasible solutions. */
    int hamCycle(int graph[][])
    {
        path = new int[V];
        for (int i = 0; i < V; i++)
            path[i] = -1;

        /* Let us put vertex 0 as the first vertex in the path.
           If there is a Hamiltonian Cycle, then the path can be
           started from any point of the cycle as the graph is
           undirected */
        path[0] = 0;
        if (hamCycleUtil(graph, path, 1) == false)
        {
            System.out.println("\nSolution does not exist");
            return 0;
        }

        printSolution(path);
        return 1;
    }

    /* A utility function to print solution */
    void printSolution(int path[])
    {
        System.out.println("Solution Exists: Following" +
                           " is one Hamiltonian Cycle");
        for (int i = 0; i < V; i++)
            System.out.print(" " + path[i] + " ");

        // Let us print the first vertex again to show the
        // complete cycle
        System.out.println(" " + path[0] + " ");
    }

    // driver program to test above function
    public static void main(String args[])
    {
        HamiltonianCycle hamiltonian =
                                new HamiltonianCycle();
        /* Let us create the following graph
           (0)--(1)--(2)
            |   / \   |
            |  /   \  |
            | /     \ |
           (3)-------(4)    */
        int graph1[][] = {{0, 1, 0, 1, 0},
            {1, 0, 1, 1, 1},
            {0, 1, 0, 0, 1},
            {1, 1, 0, 0, 1},
            {0, 1, 1, 1, 0},
        };

        // Print the solution
        hamiltonian.hamCycle(graph1);

        /* Let us create the following graph
           (0)--(1)--(2)
            |   / \   |
            |  /   \  |
            | /     \ |
           (3)       (4)    */

           System.out.println("For Graph 2:");
        int graph2[][] = {{0, 1, 0, 1, 0},
            {1, 0, 1, 1, 1},
            {0, 1, 0, 0, 1},
            {1, 1, 0, 0, 0},
            {0, 1, 1, 0, 0},
        };

        // Print the solution
        hamiltonian.hamCycle(graph2);
    }
}


/*   OUTPUT
Solution Exists: Following is one Hamiltonian Cycle
 0  1  2  4  3  0
For Graph 2:

Solution does not exist
*/
```

Pgm 8:Implement LCM algorithm

Transform and Conquer Approach

The **Transform and Conquer** technique simplifies a problem by transforming it into a more manageable form, solving the transformed problem, and then converting the solution back to the original context.

For computing the Least Common Multiple (LCM)**,** we transform the problem using the Greatest Common Divisor (GCD)**,** which can be efficiently computed using Euclidean Algorithm**.**



LCM Definition and Relation to GCD

For two numbers a and b, the **LCM** is the smallest positive integer divisible by both. The relationship between **LCM and GCD** is:

$$\text{LCM}(a,b) = \frac{|a \times b|}{\text{GCD}(a,b)}$$

Since the GCD can be found efficiently using the **Euclidean Algorithm**, this transformation makes LCM computation much faster.

Algorithm: LCM Using Transform and Conquer

Steps:

1. Compute **GCD(a, b)** using Euclidean Algorithm.
2. Compute **LCM(a, b)** using the formula:

$$\text{LCM}(a,b) = \frac{|a \times b|}{\text{GCD}(a,b)}$$

Time Complexity Analysis

- **GCD Calculation (Euclidean Algorithm):**
  - Runs in $O(\log \min(a,b))$ time.
- **LCM Calculation:**
  - Multiplication takes $O(1)$.
  - Division takes $O(1)$.
- **Overall Time Complexity:**

$$O(\log \min(a,b))$$

```
public class LCMCalculator {
    private static int gcd(int a, int b) {
        if (b == 0)
            return a;
        return gcd(b, a % b);
    }

    // Function to compute LCM of two numbers
    private static int lcm(int a, int b) {
        return (a * b) / gcd(a, b);
    }
    // Function to compute LCM of an array using transfer and conquer
    public static int lcmArray(int[] arr) {
        int result = arr[0]; // Start with the first element
        for (int i = 1; i < arr.length; i++) {
            // Transform: Combine each element with the result to form the
next state
            result = lcm(result, arr[i]);
        }
        return result;
    }

    public static void main(String[] args) {
        int[] numbers = {12, 15, 20, 25};
        int result = lcmArray(numbers);
        System.out.println("LCM of the array is: " + result);
    }

}

/* OUTPUT
 LCM of the array is: 300
 */
```