**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:



* 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:

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

### *****Algorithm: Recursive Factorial Computation*****

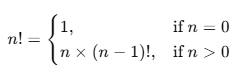
The factorial of a non-negative integer n, denoted as **n**!, is defined as:



With the base case:



Using recursion, the factorial function can be defined as:



### *****Recursive Algorithm*****

**Factorial(n)**

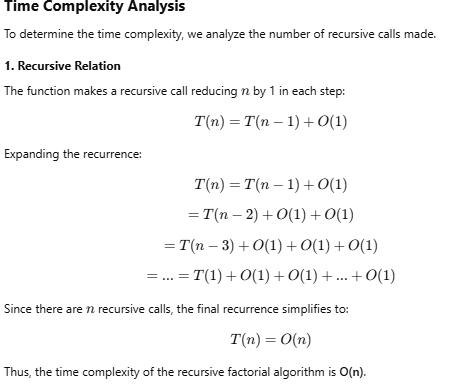
**Input: A non-negative integer n**

**Output: Factorial of n**

**1. if n == 0 then**

**2. return 1**

**3. else**

**4. return n \* Factorial(n - 1)**

***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 {

    public static long factorial(int n) {

        if (n == 0) {

            return 1;

        }

        return n \* factorial(n - 1);

    }

    public static void main(String[] args) {

        int num = 5;

        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 (Ω(n²))**

* 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. 

#### **2. Worst Case (O(n²))**

* 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:



#### **3. Average Case (Θ(n²))**

* 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:

### 

**Time Complexities:**

### *Program:*

### public class SelectionSort {

### 

### public static void selectionSort(int[] arr) {

### int n = arr.length;

### 

### 

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

### 

### int minIndex = i;

### 

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

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

### minIndex = j;

### }

### }

### swap(arr, i, minIndex);

### }

### }

### 

### public static void swap(int[] arr, int i, int j) {

### int temp = arr[i];

### arr[i] = arr[j];

### arr[j] = temp;

### }

### public static void printArray(int[] arr) {

### for (int num : arr) {

### System.out.print(num + " ");

### }

### System.out.println();

### }

### public static void main(String[] args) {

### int[] arr = {64, 25, 12, 22, 11};

### 

### System.out.println("Unsorted Array:");

### printArray(arr);

### selectionSort(arr);

### 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.

### 

### 

### *Program:*

### public class EuclidGCD {

### 

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

### 

### if (b == 0) {

### return a;

### }

### 

### return euclidGCD(b, a % b);

### }

### public static void main(String[] args) {

### 

### int a = 56;

### int b = 98;

### 

### int gcd = euclidGCD(a, b);

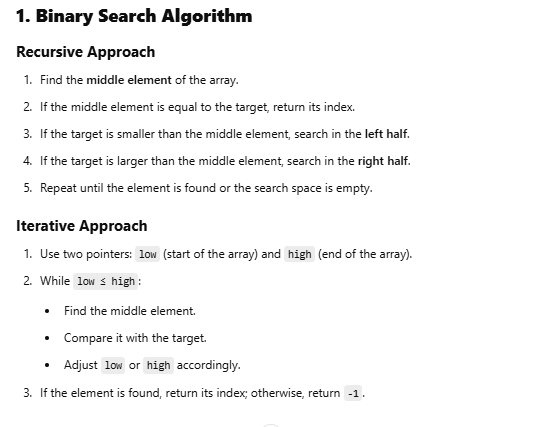
### System.out.println("The GCD of " + a + " and " + b + " is: " + gcd);

### }

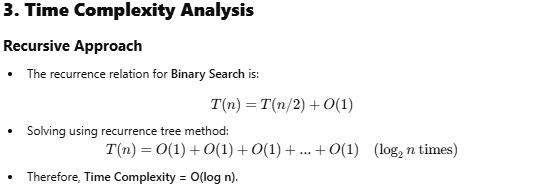
### }

### ****3.Decrease-and-Conquer Method****

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



**Time Complexity Analysis**



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));

}

}

### 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:

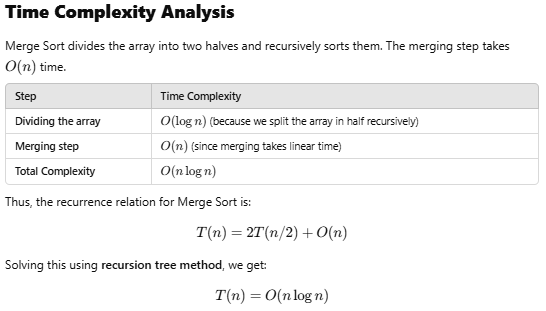
9. A[k] = rightSubArray[j]

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



**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));

}

}

### 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:

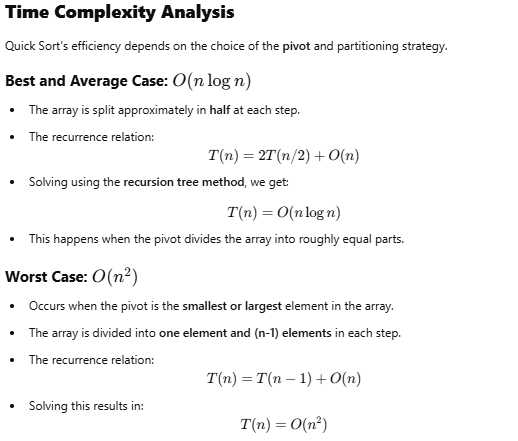
If A[j] ≤ pivot then:

i ← i + 1

Swap A[i] and A[j]

Swap A[i + 1] and A[high]

Return i + 1 // Pivot index



**Program:**

public class QuickSort {

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

public static 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, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

public static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

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);

}

public static void printArray(int[] arr) {

for (int num : arr) {

System.out.print(num + " ");

}

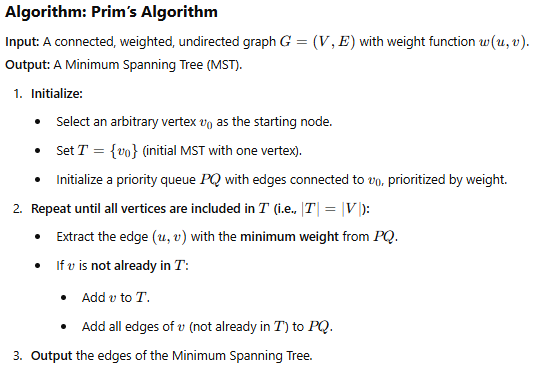
System.out.println();

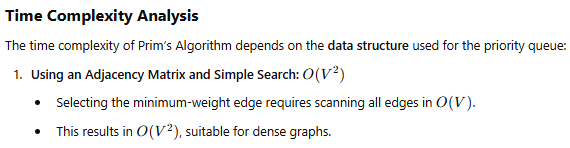
}

}

### ****5.Greedy Method****

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





***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;

        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;

            }

        }

        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;

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

                if (visited[i] == 0 && distance[i] < minDistance) {

                    v = i;

                    minDistance = distance[i];

                }

            }

            int u = from[v];

            spanning[u][v] = distance[v];

            spanning[v][u] = distance[v];

            noOfEdges--;

            visited[v] = 1;

            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

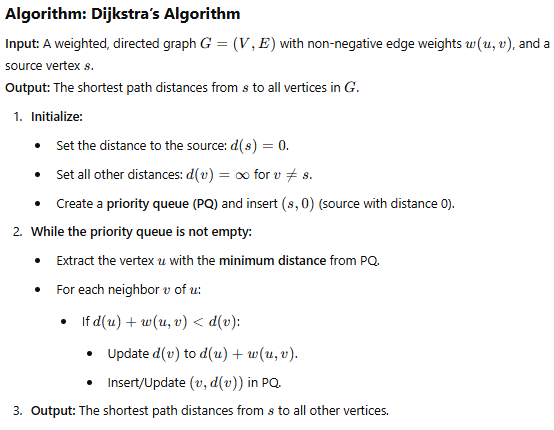
Total cost of the spanning tree = 2

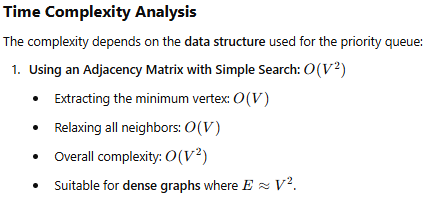
 \*/

### 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**.





### *Program:*

public class DijkstraAlgorithm {

          public void dijkstraAlgorithm(int[][] graph, int source) {

          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;

          }

          dist[source] = 0;

          for (int i = 0; i < nodes; i++) {

            int u = find\_min\_distance(dist, visited\_vertex);

            visited\_vertex[u] = true;

            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]));

          }

        }

        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) {

          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.

### 

### 

### 

### 

### *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

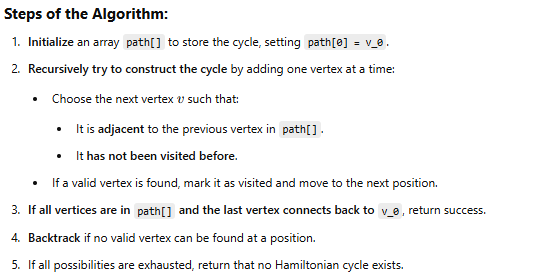
INF INF INF 0  \*/

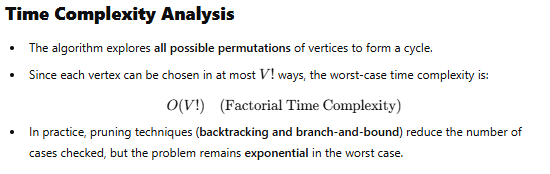
### ****Backtracking****

### 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.

### 





***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

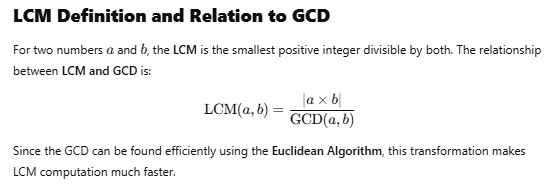
\*/

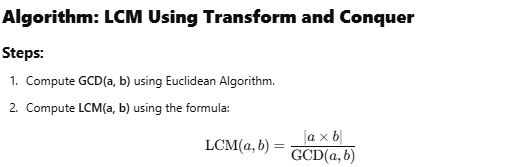
**Pgm 8:Implement LCM algorithm**

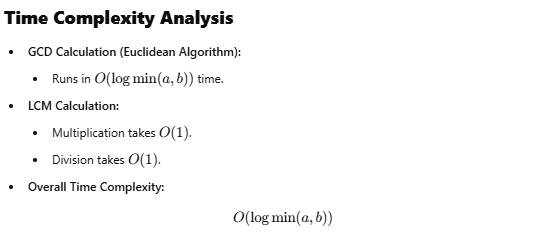
### ****Transform and Conquer Approach****

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.**







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

 \*/