1.Design and implement a sorting algorithm using Merge Sort to efficiently arrange customer orders based on their timestamps. The solution should handle a large dataset (up to 1 million orders) with minimal computational overhead. Additionally, analyze the time complexity and compare it with traditional sorting techniques.

```
import java.util.*;

class CustomerOrder {
    long timestamp;
    String orderId;

    public CustomerOrder(long timestamp, String orderId) {
        this.timestamp = timestamp;
        this.orderId = orderId;
    }

    @Override
    public String toString() {
        return "OrderID: " + orderId + ", Timestamp: " + timestamp;
    }
}

public class MergeSortOrders {
    public static void mergeSort(CustomerOrder[] orders, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(orders, left, mid);
            mergeSort(orders, mid + 1, right);
            merge(orders, left, mid, right);
        }
    }

    private static void merge(CustomerOrder[] orders, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        CustomerOrder[] L = new CustomerOrder[n1];
        CustomerOrder[] R = new CustomerOrder[n2];

        for (int i = 0; i < n1; ++i)
            L[i] = orders[left + i];
        for (int j = 0; j < n2; ++j)
            R[j] = orders[mid + 1 + j];

        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (L[i].timestamp <= R[j].timestamp) {
                orders[k] = L[i];
                i++;
            } else {
                orders[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            orders[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            orders[k] = R[j];
            j++;
            k++;
        }
    }

    public static void main(String[] args) {
        int n = 10;
        CustomerOrder[] orders = new CustomerOrder[n];
        Random rand = new Random();

        for (int i = 0; i < n; i++) {
            long ts = rand.nextInt(1_000_000);
            orders[i] = new CustomerOrder(ts, "ORD" + (i + 1));
        }

        System.out.println("Before Sorting:");
        for (int i = 0; i < Math.min(10, n); i++)
            System.out.println(orders[i]);

        long start = System.currentTimeMillis();
        mergeSort(orders, 0, orders.length - 1);
        long end = System.currentTimeMillis();

        System.out.println("\nAfter Sorting:");
        for (int i = 0; i < Math.min(10, n); i++)
            System.out.println(orders[i]);

        System.out.println("\nTime taken: " + (end - start) + " ms");
    }
}
```

2. **Movie Recommendation System Optimization A popular OTT platform, StreamFlix, offers personalized recommendations by sorting movies based on user preferences, such as IMDB rating, release year, or watch time popularity. However, during peak hours, sorting large datasets slows down the system. As a backend engineer, you must: ● Implement Quicksort to efficiently sort movies based on various user-selected parameters. ● Handle large datasets containing of movies while maintaining fast response times**

```
import java.io.*;
import java.util.*;

class Movie {
    String title;
    double imdbRating;
    int releaseYear;
    int watchTimePopularity;

    public Movie(String title, double imdbRating, int releaseYear, int watchTimePopularity) {
        this.title = title;
        this.imdbRating = imdbRating;
        this.releaseYear = releaseYear;
        this.watchTimePopularity = watchTimePopularity;
    }

    @Override
    public String toString() {
         return title + " | Rating: " + imdbRating + " | Year: " + releaseYear + " | Popularity: " +
      watchTimePopularity;
    }
}

public class MovieRecommendationSystem {

    public static void quickSort(Movie[] movies, int low, int high, String parameter) {
        if (low < high) {
            int pi = partition(movies, low, high, parameter);
            quickSort(movies, low, pi - 1, parameter);
            quickSort(movies, pi + 1, high, parameter);
        }
    }

    private static int partition(Movie[] movies, int low, int high, String parameter) {
        Movie pivot = movies[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            boolean condition = false;
            switch (parameter.toLowerCase()) {
                case "rating":
                    condition = movies[j].imdbRating > pivot.imdbRating;
                    break;
                case "year":
                    condition = movies[j].releaseYear > pivot.releaseYear;
                    break;
                case "popularity":
                    condition = movies[j].watchTimePopularity > pivot.watchTimePopularity;
                    break;
            }
            if (condition) {
                i++;
                Movie temp = movies[i];
                movies[i] = movies[j];
                movies[j] = temp;
            }
        }

        Movie temp = movies[i + 1];
        movies[i + 1] = movies[high];
        movies[high] = temp;
        return i + 1;
    }

    public static Movie[] readMoviesFromCSV(String filePath) {
        List<Movie> movies = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line = br.readLine(); // skip header
            while ((line = br.readLine()) != null) {
                String[] values = line.split(",");
                if (values.length < 4) continue;
                try {
                    String title = values[0].trim();
                    double rating = Double.parseDouble(values[1].trim());
                    int year = Integer.parseInt(values[2].trim());
                    int popularity = Integer.parseInt(values[3].trim());
                    movies.add(new Movie(title, rating, year, popularity));
                } catch (Exception e) {
                    // skip invalid rows
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }

        return movies.toArray(new Movie[0]);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter CSV file path (e.g., movies.csv): ");
        String filePath = sc.nextLine();

        Movie[] movies = readMoviesFromCSV(filePath);

        if (movies.length == 0) {
            System.out.println("No movie data found in the file.");
            return;
        }

        System.out.println("Sort by (rating/year/popularity): ");
        String parameter = sc.next();

        long start = System.currentTimeMillis();
        quickSort(movies, 0, movies.length - 1, parameter);
        long end = System.currentTimeMillis();

        System.out.println("\nTop 10 Sorted Movies by " + parameter + ":");
        for (int i = 0; i < Math.min(10, movies.length); i++)
            System.out.println(movies[i]);

        System.out.println("\nTotal Movies: " + movies.length);
        System.out.println("Time taken: " + (end - start) + " ms");
    }
}
```

3. A devastating flood has hit multiple villages in a remote area, and the government, along with NGOs, is organizing an emergency relief operation. A rescue team has a limited-capacity boat that can carry a maximum weight of W kilograms. The boat must transport critical supplies, including food, medicine, and drinking water, from a relief center to the affected villages. Each type of relief item has: ● A weight (wi) in kilograms. ● Utility value (vi) indicating its importance (e.g., medicine has higher value than food). ● Some items can be divided into smaller portions (e.g., food and water), while others must be taken as a whole (e.g., medical kits).

```
import java.util.*;

class Item {
    String name;
    double weight, value;
    boolean divisible;
    int priority; // 1=High, 2=Medium, 3=Low

    Item(String n, double w, double v, boolean d, int p) {
        name = n; weight = w; value = v; divisible = d; priority = p;
    }

    double ratio() { return value / weight; }
}

public class FractionalKnapsack {

    static double knapsack(List<Item> items, double capacity) {
        // Sort by priority, then by value/weight ratio
        items.sort((a, b) -> {
            if (a.priority != b.priority) return a.priority - b.priority;
            return Double.compare(b.ratio(), a.ratio());
        });

        double totalValue = 0, totalWeight = 0;
        System.out.println("\nSelected items:");

        for (Item it : items) {
            if (capacity <= 0) break;

            double take = 0;
            if (it.divisible) { // partial allowed
                take = Math.min(it.weight, capacity);
            } else if (it.weight <= capacity) { // full only
                take = it.weight;
            }

            if (take > 0) {
                double val = it.ratio() * take;
                totalValue += val;
                totalWeight += take;
                capacity -= take;

                System.out.println(" - " + it.name +
                        " | Taken: " + take + "kg" +
                        " | Value: " + val +
                        " | Priority: " + it.priority +
                        " | Type: " + (it.divisible ? "Divisible" : "Indivisible"));
            }
        }

        System.out.printf("\nTotal weight: %.2f kg\nTotal value: %.2f\n",
                totalWeight, totalValue);
        return totalValue;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of items: ");
        int n = sc.nextInt();

        List<Item> items = new ArrayList<>();
        for (int i = 1; i <= n; i++) {
            System.out.println("\nItem " + i + ":");
            System.out.print("Name: "); String name = sc.next();
            System.out.print("Weight: "); double w = sc.nextDouble();
            System.out.print("Value: "); double v = sc.nextDouble();
            System.out.print("Divisible (1/0): "); boolean d = sc.nextInt() == 1;
            System.out.print("Priority (1=High,2=Med,3=Low): "); int p = sc.nextInt();
            items.add(new Item(name, w, v, d, p));
        }

        System.out.print("\nEnter max capacity: ");
        double cap = sc.nextDouble();

        knapsack(items, cap);
        sc.close();
    }
}
```

4. In a modern smart city infrastructure, efficient emergency response is critical to saving lives. One major challenge is ensuring that ambulances reach hospitals in the shortest possible time, especially during peak traffic conditions. The city&#39;s road network can be modeled as a graph, where intersections are represented as nodes and roads as edges with weights indicating real-time travel time based on current traffic congestion. The goal is to design and implement an intelligent traffic management system that dynamically computes the fastest route from an ambulance’s current location (source node S) to the nearest hospital (destination node D) in the network. Due to ever-changing traffic conditions, edge weights must be updated in real-time, requiring the system to adapt and re-compute optimal routes as necessary.

```
import java.util.*;

public class DijkstraAmbulance {
    static class Edge {
        int to, w;
        Edge(int t, int w) { this.to = t; this.w = w; }
    }

    static void dijkstra(int src, List<List<Edge>> g, int[] dist, int[] par) {
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0; Arrays.fill(par, -1);

        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
        pq.add(new int[]{src, 0});

        while (!pq.isEmpty()) {
            int[] cur = pq.poll();
            int u = cur[0], d = cur[1];
            if (d > dist[u]) continue;

            for (Edge e : g.get(u)) {
                if (dist[e.to] > d + e.w) {
                    dist[e.to] = d + e.w;
                    par[e.to] = u;
                    pq.add(new int[]{e.to, dist[e.to]});
                }
            }
        }
    }

    static List<Integer> path(int dest, int[] par) {
        List<Integer> p = new ArrayList<>();
        for (int i = dest; i != -1; i = par[i]) p.add(i);
        Collections.reverse(p);
        return p;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter vertices and edges: ");
        int V = sc.nextInt(), E = sc.nextInt();

        List<List<Edge>> g = new ArrayList<>();
        for (int i = 0; i < V; i++) g.add(new ArrayList<>());

        System.out.println("Enter edges (u v w):");
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt(), v = sc.nextInt(), w = sc.nextInt();
            g.get(u).add(new Edge(v, w));
            g.get(v).add(new Edge(u, w)); // bidirectional
        }

        System.out.print("Enter ambulance start: ");
        int src = sc.nextInt();
        System.out.print("Enter hospital count: ");
        int h = sc.nextInt();
        int[] hosp = new int[h];
        System.out.print("Hospitals: ");
        for (int i = 0; i < h; i++) hosp[i] = sc.nextInt();

        int[] dist = new int[V], par = new int[V];
        dijkstra(src, g, dist, par);

        int nearest = -1, min = Integer.MAX_VALUE;
        for (int x : hosp) if (dist[x] < min) { min = dist[x]; nearest = x; }

        System.out.println("\nNearest hospital: " + nearest + " (" + min + " min)");
        System.out.println("Path: " + path(nearest, par));
        sc.close();
    }
}
```

```
Enter vertices and edges: 4
4
Enter edges (u v w):
0 1 5
0 2 10
1 3 3
2 3 1
Enter ambulance start: 0
Enter hospital count: 1
Hospitals: 2

Nearest hospital: 2 (9 min)
Path: [0, 1, 3, 2]
```

5. A logistics company, SwiftCargo, specializes in delivering packages across multiple cities. To optimize its delivery process, the company divides the transportation network into multiple stages (warehouses, transit hubs, and final delivery points). Each package must follow the most cost- efficient or fastest route from the source to the destination while passing through these predefined stages. As a logistics optimization engineer, you must: 1. Model the transportation network as a directed, weighted multistage graph with multiple intermediate stages. 2. Implement an efficient algorithm (such as Dynamic Programming or Dijkstra's Algorithm) to find the optimal delivery route. 3. Ensure that the algorithm scales for large datasets (handling thousands of cities and routes). 4. Analyze and optimize route selection based on real-time constraints, such as traffic conditions, weather delays, or fuel efficiency. Constraints &amp; Considerations: ● The network is structured into N stages, and every package must pass through at least one node in each stage. ● There may be multiple paths with different costs/times between stages. ● The algorithm should be flexible enough to handle real-time changes (e.g., road closures or rerouting requirements). ● The system should support batch processing for multiple delivery requests Tasks: 1. Model the network as a directed weighted

multistage graph. 2. Use Dynamic Programming (DP) to compute optimal delivery routes. 3. Ensure scalability for thousands of cities/routes. 4. Consider real-time constraints like traffic, weather, or fuel efficiency. Constraints: 1. Each package must pass through at least one node per stage. 2. Multiple paths may exist between stages. 3. Support batch processing and dynamic rerouting.

```
import java.util.*;

public class SwiftCargoRouting{

   static class Edge{
      int from,to,cost;

      Edge(int f,int t,int c){ from=f; to=t; cost=c;}
   }

   static int[] stage;
   static List<Edge>[] graph;
   static int [] dist,parent;


   static void shortestPath(int V,int stages){
      Arrays.fill(dist, Integer.MAX_VALUE);
      dist[0]=0;

      for(int s=0;s<stages;s++){
         for(int u=0;u<V;u++){
            if(stage[u]!=s || dist[u]==Integer.MAX_VALUE) continue;

            for(Edge e:graph[u]){
               if(dist[e.to]>dist[u]+e.cost){
                  dist[e.to]=dist[u]+e.cost;
                  parent[e.to]=u;

               }
            }
         }
      }
   }

   static List<Integer> getPath(int dest){

      List<Integer> path=new ArrayList<>();

      for(int i=dest;i!=-1;i=parent[i]) {
         path.add(i);
      }
      Collections.reverse(path);
      return path;
   }

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

```
        System.out.print("Enter the no total of cities:");
        int V=sc.nextInt();

        System.out.print("Enter the no of stages:");
        int stages=sc.nextInt();

        stage=new int[V];

        System.out.println("Enter stage of each city (0 to"+(stages-1)+"):");

        for(int i=0;i<V;i++) stage[i]=sc.nextInt();

        @SuppressWarnings("unchecked")
        List<Edge>[] temp = new ArrayList[V];
        graph = temp;

        for(int i=0;i<V;i++){
            graph[i]=new ArrayList<>();
        }

        System.out.println("Enter the total routes(edge):");
        int E=sc.nextInt();
        System.out.println("Enter the each route from to cost:");
        for (int i=0;i<E;i++){
            int u=sc.nextInt();
            int v=sc.nextInt();
            int c=sc.nextInt();

            if(stage[v]>stage[u]){
                graph[u].add(new Edge(u,v,c));
            }
        }

        dist=new int[V];
        parent=new int[V];

        Arrays.fill(parent,-1);

        shortestPath(V, stages);

        System.out.println("Enter the destination city:");
        int dest=sc.nextInt();

        System.err.println("Minimum Cost:"+dist[dest]);
        System.err.println("Optimal Path:"+getPath(dest));

        sc.close();
    }
}
```

```
Enter the no total of cities:4
Enter the no of stages:2
Enter stage of each city (0 to1):
```

```
0 0 1 1
Enter the total routes(edge):
3
Enter the each route from to cost:
0 2 4
0 3 3
1 2 2
Enter the destination city:
3
Minimum Cost:3
Optimal Path:[0, 3]
```

6. A massive earthquake has struck a remote region, and a relief organization is transporting essential supplies to the affected area. The organization has a limited-capacity relief truck that can carry a maximum weight of W kg. They have N different types of essential items, each with a specific weight and an associated utility value (importance in saving lives and meeting urgent needs). Since the truck has limited capacity, you must decide which items to include to maximize the total utility value while ensuring the total weight does not exceed the truck&#39;s limit. Your Task as a Logistics Coordinator: 1. Model this problem using the 0/1 Knapsack approach, where each item can either be included in the truck (1) or not (0). 2. Implement an algorithm to find the optimal set of items that maximizes utility while staying within the weight constraint. 3. Analyze the performance of different approaches (e.g., Brute Force, Dynamic Programming, and Greedy Algorithms) for solving this problem efficiently. 4. Optimize for real-world constraints, such as perishable items (medicines, food) having priority over less critical supplies. Extend the model to consider multiple trucks or real-time decision- making for dynamic supply chain management.

```java
import java.util.*;

public class DisasterRelief01 {

    static class Item {
        String name;
        int weight, value, priority; // smaller priority number = higher priority
        Item(String n, int w, int v, int p) { name = n; weight = w; value = v; priority = p; }
    }

    // 0/1 Knapsack using Dynamic Programming
    static int[][] knapsackDP(List<Item> items, int W) {
        int n = items.size();
        int[][] dp = new int[n + 1][W + 1];
        for (int i = 1; i <= n; i++) {
            Item it = items.get(i - 1);
            for (int w = 0; w <= W; w++) {
                if (it.weight <= w)
                    dp[i][w] = Math.max(it.value + dp[i - 1][w - it.weight], dp[i - 1][w]);
                else dp[i][w] = dp[i - 1][w];
            }
        }
        return dp;
    }
    static List<Item> getSelectedItems(List<Item> items, int W, int[][] dp) {
        List<Item> selected = new ArrayList<>();
        int n = items.size(), w = W;
        for (int i = n; i > 0; i--) {
            if (dp[i][w] != dp[i - 1][w]) {
                Item it = items.get(i - 1);
                selected.add(it);
                w -= it.weight;
            }
        }
        Collections.reverse(selected);
        return selected;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of items: ");
        int n = sc.nextInt();

        List<Item> items = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            System.out.println("\nItem #" + (i + 1));
            System.out.print("Name: "); String name = sc.next();
            System.out.print("Weight (kg): "); int weight = sc.nextInt();
            System.out.print("Utility Value: "); int value = sc.nextInt();
            System.out.print("Priority (1=High, 2=Medium, 3=Low): "); int p = sc.nextInt();
            items.add(new Item(name, weight, value, p));
        }

        System.out.print("\nEnter truck capacity (W in kg): ");
        int W = sc.nextInt();

        // Sort by priority before applying knapsack
        items.sort(Comparator.comparingInt(it -> it.priority));

        int[][] dp = knapsackDP(items, W);
        List<Item> selected = getSelectedItems(items, W, dp);

        int totalValue = dp[items.size()][W];
        int totalWeight = selected.stream().mapToInt(i -> i.weight).sum();

        System.out.println("\n===== Optimal Resource Allocation =====");
        for (Item it : selected)
                System.out.println(it.name + " | Weight: " + it.weight + " | Value: " + it.value + " |
Priority: " + it.priority);
        System.out.println("------------------------------------");
        System.out.println("Total Weight: " + totalWeight + " / " + W);
        System.out.println("Total Utility: " + totalValue);

        sc.close();
    }
}
```

```
Enter truck capacity (W in kg):

60

===== Optimal Resource Allocation =====
med | Weight: 10 | Value: 100 | Priority: 1
firstaid | Weight: 40 | Value: 80 | Priority: 1
-----------------------------------
Total Weight: 50 / 60
Total Utility: 180
```

7. Scenario: University Timetable Scheduling A university is facing challenges in scheduling exam timetables due to overlapping student enrolments in multiple courses. To prevent clashes, the university needs to assign exam slots efficiently, ensuring that no two exams taken by the same student are scheduled at the same time. To solve this, the university decides to model the problem as a Graph Colouring Problem, where: Each course is represented as a vertex. An edge exists between two vertices if a student is enrolled in both courses. Each vertex (course) must be assigned a colour (time slot) such that no two adjacent vertices share the same colour (no two exams with common students are scheduled in the same slot). As a scheduling system developer, you must: 1. Model the problem as a graph and implement a graph colouring algorithm (e.g., Greedy Colouring or Backtracking). 2. Minimize the number of colours (exam slots) needed while ensuring conflict-free scheduling. 3. Handle large datasets with thousands of courses and students, optimizing performance. 4. Compare the efficiency of Greedy Colouring, DSATUR, and Welsh-Powell algorithms for better scheduling. Extend the solution to include room allocation constraints where exams in the same slot should fit within available classrooms.

```java
import java.util.*;

public class TTScheduling {

    static int V; // number of courses
    static int[][] graph; // adjacency matrix
    static int[] color; // color assignment
    static int totalColors; // total available slots

    // Check if the current color can be assigned to course v
    static boolean isSafe(int v, int c) {
        for (int i = 0; i < V; i++)
            if (graph[v][i] == 1 && color[i] == c) // adjacent course with same color
                return false;
        return true;
    }
```

```
// Backtracking function
static boolean solveGraphColoring(int v) {
    if (v == V) return true; // all courses colored successfully

    for (int c = 1; c <= totalColors; c++) {
        if (isSafe(v, c)) {
            color[v] = c;
            if (solveGraphColoring(v + 1)) return true;
            color[v] = 0; // backtrack
        }
    }
    return false;
}

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

    System.out.print("Enter number of courses: ");
    V = sc.nextInt();
    graph = new int[V][V];

    System.out.print("Enter number of conflict pairs (edges): ");
    int E = sc.nextInt();

    System.out.println("Enter conflict pairs (u v): ");
    for (int i = 0; i < E; i++) {
        int u = sc.nextInt(), v = sc.nextInt();
        graph[u][v] = graph[v][u] = 1;
    }

    System.out.print("Enter maximum number of available exam slots: ");
    totalColors = sc.nextInt();

    color = new int[V];
    Arrays.fill(color, 0);

    if (solveGraphColoring(0)) {
        System.out.println("\n===== Exam Timetable (Course → Slot) =====");
        for (int i = 0; i < V; i++)
            System.out.println("Course " + i + " → Slot " + color[i]);
        System.out.println("\nTotal Slots Used: " + Arrays.stream(color).max().getAsInt());
    } else {
        System.out.println("No valid scheduling possible with given slots.");
    }

    // --- Optional Room Allocation ---
    System.out.print("\nEnter number of classrooms available per slot: ");
    int rooms = sc.nextInt();
    int maxSlot = Arrays.stream(color).max().getAsInt();
    if (rooms < maxSlot)
        System.out.println("Not enough rooms for all exam slots!");
    else
        System.out.println("Room allocation possible for all slots.");
        sc.close();
    }
}
```

```
Enter number of courses: 4
Enter number of conflict pairs (edges): 3
Enter conflict pairs (u v):
0 1
1 2
2 3
Enter maximum number of available exam slots: 3

===== Exam Timetable (Course → Slot) =====
Course 0 → Slot 1
Course 1 → Slot 2
Course 2 → Slot 1
Course 3 → Slot 2

Total Slots Used: 2

Enter number of classrooms available per slot: 3
Room allocation possible for all slots.
```

8. A leading logistics company, SwiftShip, is responsible for delivering packages to multiple cities. To minimize fuel costs and delivery time, the company needs to find the shortest possible route that allows a delivery truck to visit each city exactly once and return to the starting point. The company wants an optimized solution that guarantees the least cost route, considering: ● Varying distances between cities. ● Fuel consumption costs, which depend on road conditions. ● Time constraints, as deliveries must be completed within a given period. Since there are N cities, a brute-force approach checking all (N-1)!permutations is infeasible for large N (e.g., 20+ cities). Therefore, you must implement an LC (Least Cost) Branch and Bound algorithm to find the optimal route while reducing unnecessary computations efficiently.

```java
import java.util.*;

public class TSP {
    static int N;
    static int[][] dist;
    static int minCost = Integer.MAX_VALUE;
    static int[] bestPath;

    static void tspBranchBound(int[] path, boolean[] visited, int level, int cost) {
        if (level == N) {
            cost += dist[path[level-1]][path[0]];
            if (cost < minCost) {
                minCost = cost;
                bestPath = path.clone();
            }
            return;
        }

        for (int i = 0; i < N; i++) {
            if (!visited[i]) {
                int newCost = cost + dist[path[level-1]][i];
                if (newCost < minCost) { // Bound condition
                    path[level] = i;
                    visited[i] = true;
                    tspBranchBound(path, visited, level + 1, newCost);
                    visited[i] = false;
                }
            }
        }
    }

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

        System.out.print("Enter number of cities: ");
        N = sc.nextInt();

        dist = new int[N][N];
        System.out.println("Enter distance matrix:");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                dist[i][j] = sc.nextInt();
            }
        }

        int[] path = new int[N];
        boolean[] visited = new boolean[N];
        path[0] = 0; // Start from city 0
        visited[0] = true;

        System.out.println("\nCalculating optimal route...");
        tspBranchBound(path, visited, 1, 0);

        System.out.println("\nOptimal Route:");
        for (int i = 0; i < N; i++) {
            System.out.print(bestPath[i] + " → ");
        }
        System.out.println(bestPath[0]);

        System.out.println("Minimum Cost: " + minCost);

        sc.close();
    }
}
```

```
Enter number of cities: 4
Enter distance matrix:
0 10  15 20
10 0  35 20
```

```
15 35 0  10
20 15 10
0

Calculating optimal route...

Optimal Route:
0 → 2 → 3 → 1 → 0
Minimum Cost: 50
```