**Problem Statement**

**Amar is tasked with creating a program that solves the Warehouse Delivery Optimization Problem. The goal is to find the shortest possible route that visits a set of warehouses and returns to the starting warehouse.**

**You are required to implement a program that takes input regarding the number of warehouses and the distance matrix between them and outputs the optimal route and its associated minimum cost using the Traveling Salesman Problem (TSP).**

**Note: The diagonal values will be 0, as the distance from a warehouse to itself is 0.**

**Input Format :**

The first line of input consists of an integer **n**, representing the number of warehouses.

The next**n** lines contain**n** space-separated integers each, representing the distance matrix between the warehouses.

The cell (i, j) represents the distance from warehouse **i** to warehouse**j.**

**Output Format :**

The first line of output displays "The Route is: " followed by the route of the delivery using ---> in between.

The second line displays "Minimum cost is " followed by the total minimum cost associated with the shortest route.

**Refer to the sample output for the formatting specifications.**

**Sample testcases :**

Testcase 1 Input

Testcase 1 Output

4

0 4 1 3

4 0 2 1

1 2 0 5

3 1 5 0

The Path is: 1--->3--->2--->4--->1

Minimum cost is 7

Testcase 2 Input

Testcase 2 Output

3

0 10 15

10 0 20

15 20 0

The Path is: 1--->2--->3--->1

Minimum cost is 45

**SOLUTION:**

// You are using Java

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.Scanner;

class DeliveryRouteOptimization {

    // Variable to keep track of the minimum cost found

    public static int minCost = Integer.MAX\_VALUE;

    // List to store the best route found with the minimum cost

    public static List<Integer> bestRoute = new ArrayList<>();

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Input number of distribution centers (nodes)

        int n = sc.nextInt();

        // Input the adjacency matrix which represents the cost between every pair of nodes

        int[][] costMatrix = new int[n][n];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                costMatrix[i][j] = sc.nextInt();

            }

        }

        // Create a list of nodes (cities) to permute

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

        for (int i = 0; i < n; i++) {

            currentRoute.add(i);

        }

        // Generate all permutations of the route and find the minimum cost

        permute(currentRoute, 0, n - 1, costMatrix);

        // Output the best route and its cost

        System.out.print("The Path is: ");

        for (int i = 0; i < bestRoute.size(); i++) {

            System.out.print((bestRoute.get(i) + 1)); // Adding 1 to adjust for 1-based index

            if (i != bestRoute.size() - 1) {

                System.out.print("--->");

            }

        }

        System.out.println();

        System.out.println("Minimum cost is " + minCost);

    }

    // Function to swap two elements in the list

    public static void swap(List<Integer> list, int i, int j) {

        int temp = list.get(i);

        list.set(i, list.get(j));

        list.set(j, temp);

    }

    // Function to generate all permutations of the list

    public static void permute(List<Integer> list, int l, int r, int[][] costMatrix) {

        // If we have a complete permutation

        if (l == r) {

            // Calculate the cost of the current permutation

            calculateCost(list, costMatrix);

        } else {

            // Generate permutations by swapping each element in the list

            for (int i = l; i <= r; i++) {

                // Swap the current element with the start element

                swap(list, l, i);

                // Recurse with the next part of the list

                permute(list, l + 1, r, costMatrix);

                // Backtrack by swapping back

                swap(list, l, i);

            }

        }

    }

    // Function to calculate the cost of a specific route and update the minimum cost

    public static void calculateCost(List<Integer> route, int[][] costMatrix) {

        int currentCost = 0;

        // Calculate the cost of traveling through the route

        for (int i = 0; i < route.size() - 1; i++) {

            currentCost += costMatrix[route.get(i)][route.get(i + 1)];

        }

        // Add the cost of returning to the starting point

        currentCost += costMatrix[route.get(route.size() - 1)][route.get(0)];

        // Update the minimum cost and best route if a better one is found

        if (currentCost < minCost || (currentCost == minCost && isLexicographicallySmaller(route, bestRoute))) {

            minCost = currentCost;

            bestRoute = new ArrayList<>(route);

            bestRoute.add(route.get(0)); // Return to the starting point

        }

    }

    // Function to compare two routes lexicographically

    public static boolean isLexicographicallySmaller(List<Integer> route1, List<Integer> route2) {

        for (int i = 0; i < route1.size(); i++) {

            if (route1.get(i) < route2.get(i)) {

                return true;

            } else if (route1.get(i) > route2.get(i)) {

                return false;

            }

        }

        return false;

    }

}

Integer.MAX\_VALUE in Java represents the maximum value that an int can hold, which is 231−12^{31} - 1231−1 or 2,147,483,647. It is often used as a sentinel value to indicate an initial or infinite maximum value in algorithms, such as when finding the minimum in a set of values.

If you're looking for alternatives to Integer.MAX\_VALUE, you have a few options depending on your needs:

**1. Long.MAX\_VALUE**

If the range of values you need exceeds what can be represented by an int, you might use Long.MAX\_VALUE instead. This is the maximum value that a long can hold, which is 263−12^{63} - 1263−1 or 9,223,372,036,854,775,807.

java

Copy code

public static long minCost = Long.MAX\_VALUE;

**2. Double.POSITIVE\_INFINITY**

For algorithms where you may need to handle floating-point numbers, Double.POSITIVE\_INFINITY represents positive infinity in the double data type. This is useful if your cost values might involve decimals or if you want to ensure that the initial value is higher than any possible real value.

java

Copy code

public static double minCost = Double.POSITIVE\_INFINITY;

Certainly! Let's delve into the line:

java

Copy code

permute(currentRoute, 0, n - 1, costMatrix);

This line is a call to the permute method, which generates all possible permutations of a list and calculates the cost for each permutation to find the minimum cost route in the Traveling Salesman Problem (TSP). Here’s a detailed breakdown of how this works:

**permute Method Breakdown**

**Method Definition:**

java

Copy code

public static void permute(List<Integer> list, int l, int r, int[][] costMatrix)

**Parameters:**

* list: The list of nodes (cities) that we want to permute.
* l: The starting index of the permutation segment.
* r: The ending index of the permutation segment (usually the last index of the list).
* costMatrix: A 2D array representing the cost between each pair of nodes.

**Process**

1. **Base Case:**

java

Copy code

if (l == r) {

calculateCost(list, costMatrix);

}

* + **Condition**: When the start index l equals the end index r, it means we have a complete permutation of the list.
  + **Action**: Call calculateCost(list, costMatrix) to calculate the cost of this permutation and update the minimum cost if this permutation is better.

1. **Recursive Case:**

java

Copy code

for (int i = l; i <= r; i++) {

swap(list, l, i);

permute(list, l + 1, r, costMatrix);

swap(list, l, i); // backtrack

}

* + **Loop**: Iterates from the start index l to the end index r.
  + **Swap**: Swaps the element at the start index l with the element at index i. This operation is crucial for generating different permutations by changing the position of the element at l.
  + **Recursive Call**: After swapping, call permute(list, l + 1, r, costMatrix) to permute the rest of the list starting from index l + 1.
  + **Backtrack**: Swap back the elements to restore the original order of the list. This ensures that the next iteration of the loop starts with the original list configuration.

**Detailed Example**

Let's assume we have a list [0, 1, 2] and n = 3. The permute method will generate all permutations of this list as follows:

1. **Initial Call:**

java

Copy code

permute([0, 1, 2], 0, 2, costMatrix);

1. **First Iteration of Loop in permute:**
   * Swap 0 (at index 0) with 0 (at index 0): [0, 1, 2]
   * Call permute([0, 1, 2], 1, 2, costMatrix)
2. **Second Level of Recursion:**
   * Swap 1 (at index 1) with 1 (at index 1): [0, 1, 2]
   * Call permute([0, 1, 2], 2, 2, costMatrix), which reaches the base case and calculates the cost.
   * Swap 1 (at index 1) back with 1 (at index 1): [0, 1, 2]
   * Swap 1 (at index 1) with 2 (at index 2): [0, 2, 1]
   * Call permute([0, 2, 1], 2, 2, costMatrix), which calculates the cost.
   * Backtrack: Swap 2 (at index 2) back with 1 (at index 1): [0, 1, 2]
3. **Back to First Level of Recursion:**
   * Swap 0 (at index 0) with 1 (at index 1): [1, 0, 2]
   * Call permute([1, 0, 2], 1, 2, costMatrix), which generates permutations [1, 0, 2] and [1, 2, 0].
   * Backtrack: Swap 1 (at index 1) back with 0 (at index 0): [0, 1, 2]
   * Swap 0 (at index 0) with 2 (at index 2): [2, 1, 0]
   * Call permute([2, 1, 0], 1, 2, costMatrix), which generates permutations [2, 1, 0] and [2, 0, 1].
   * Backtrack: Swap 2 (at index 2) back with 0 (at index 0): [0, 1, 2]

**Summary**

The permute method uses recursion and backtracking to generate all possible permutations of the list. For each permutation, it calculates the cost of the route and updates the minimum cost and best route if a better one is found. This approach ensures that every possible route is considered, though it becomes computationally expensive as the number of nodes increases.

Certainly! Let’s break down the calculateCost method and understand each part in detail:

**calculateCost Method**

**Method Definition:**

java

Copy code

public static void calculateCost(List<Integer> route, int[][] costMatrix) {

int currentCost = 0;

// Calculate the cost of traveling through the route

for (int i = 0; i < route.size() - 1; i++) {

currentCost += costMatrix[route.get(i)][route.get(i + 1)];

}

// Add the cost of returning to the starting point

currentCost += costMatrix[route.get(route.size() - 1)][route.get(0)];

// Update the minimum cost and best route if a better one is found

if (currentCost < minCost || (currentCost == minCost && isLexicographicallySmaller(route, bestRoute))) {

minCost = currentCost;

bestRoute = new ArrayList<>(route);

bestRoute.add(route.get(0)); // Return to the starting point

}

}

**Detailed Explanation**

1. **Initialization:**

java

Copy code

int currentCost = 0;

* + **Purpose**: Initialize currentCost to store the total cost of the current route permutation.

1. **Calculate the Cost of the Route:**

java

Copy code

for (int i = 0; i < route.size() - 1; i++) {

currentCost += costMatrix[route.get(i)][route.get(i + 1)];

}

* + **Loop**: Iterates through the route list to sum the cost between consecutive cities.
  + **Index i**: Represents the current city in the route.
  + **Index i + 1**: Represents the next city in the route.
  + **costMatrix[route.get(i)][route.get(i + 1)]**: Retrieves the travel cost between the current city and the next city from the costMatrix (adjacency matrix).

1. **Add the Cost of Returning to the Starting Point:**

java

Copy code

currentCost += costMatrix[route.get(route.size() - 1)][route.get(0)];

* + **Purpose**: Adds the cost to return from the last city in the route back to the starting city, completing the tour.

1. **Update Minimum Cost and Best Route:**

java

Copy code

if (currentCost < minCost || (currentCost == minCost && isLexicographicallySmaller(route, bestRoute))) {

minCost = currentCost;

bestRoute = new ArrayList<>(route);

bestRoute.add(route.get(0)); // Return to the starting point

}

* + **Condition**:
    - **currentCost < minCost**: Checks if the cost of the current route is less than the previously recorded minimum cost. If true, update the minimum cost and best route.
    - **currentCost == minCost && isLexicographicallySmaller(route, bestRoute)**: If the cost is the same as the current minimum but the route is lexicographically smaller (i.e., comes first in dictionary order), update the best route. This ensures that among routes with the same cost, the smallest lexicographical route is chosen.
  + **Update**:
    - **minCost = currentCost**: Update the minimum cost found.
    - **bestRoute = new ArrayList<>(route)**: Update the bestRoute to the current route.
    - **bestRoute.add(route.get(0))**: Add the starting city to the end of the route to complete the circular tour.

**Summary**

The calculateCost method:

* Computes the total cost of traveling through a specific route by summing the travel costs between consecutive cities and returning to the starting point.
* Updates the global minCost and bestRoute if the current route’s cost is lower or if the cost is the same but the route is lexicographically smaller.

This method is crucial for determining the optimal route by evaluating each permutation’s cost and ensuring that the best (cheapest) route is selected.

The isLexicographicallySmaller method is used to compare two routes in a lexicographical order. Lexicographical order is similar to alphabetical order but applied to numbers. Here's a detailed explanation of how it works:

**isLexicographicallySmaller Method**

**Method Definition:**

java

Copy code

public static boolean isLexicographicallySmaller(List<Integer> route1, List<Integer> route2) {

for (int i = 0; i < route1.size(); i++) {

if (route1.get(i) < route2.get(i)) {

return true;

} else if (route1.get(i) > route2.get(i)) {

return false;

}

}

return false;

}

**Detailed Explanation**

1. **Loop Through the Routes:**

java

Copy code

for (int i = 0; i < route1.size(); i++) {

* + **Purpose**: Iterate through each element of both routes. Since both routes should be of the same size, the loop runs for each index of the routes.

1. **Compare Elements at Each Index:**

java

Copy code

if (route1.get(i) < route2.get(i)) {

return true;

} else if (route1.get(i) > route2.get(i)) {

return false;

}

* + **route1.get(i) < route2.get(i)**: If the element at index i in route1 is less than the corresponding element in route2, then route1 is lexicographically smaller, so return true.
  + **route1.get(i) > route2.get(i)**: If the element at index i in route1 is greater than the corresponding element in route2, then route1 is not smaller, so return false.

1. **Return false if All Elements are Equal:**

java

Copy code

return false;

* + **Purpose**: If none of the elements at the corresponding indices are different, the two routes are lexicographically equal. Therefore, route1 is not considered smaller than route2, so return false.

**Lexicographical Order**

Lexicographical order means comparing two sequences in a manner similar to dictionary order. For numbers, this means:

* **Smaller**: [1, 2, 3] is lexicographically smaller than [1, 3, 2] because at the first index where they differ, 2 (from [1, 2, 3]) is less than 3 (from [1, 3, 2]).

**Example**

Assume we have two routes:

* route1 = [2, 1, 3]
* route2 = [2, 2, 1]

**Comparison Steps:**

1. **Index 0**: Both elements are 2. Continue to next index.
2. **Index 1**: route1.get(1) is 1, and route2.get(1) is 2. Since 1 < 2, route1 is lexicographically smaller than route2.

Therefore, the method will return true for these routes.

**Summary**

The isLexicographicallySmaller method compares two routes in a manner similar to dictionary order. It ensures that among routes with the same cost, the route that comes first in lexicographical order is chosen as the best route. This is useful for ensuring consistency in route selection, especially when multiple routes have the same cost.

**Problem Statement**

**You are a delivery manager who needs to plan the delivery route for a fleet of trucks to a set of distribution centers in a city. Each distribution center is at a different location, and the cost of traveling between centers is known.**

**Your goal is to find the shortest route that allows a truck to visit each distribution center exactly once and return to the starting point, using the Traveling Salesman Problem (TSP). Write a program to calculate the minimum cost of this route and output the optimal path.**

**Input Format :**

The first line contains an integer **n**, the number of distribution centers.

The next **n** lines contain **n** space-separated integers each, representing the adjacency matrix.

The ith integer on the jth line represents the cost of traveling from distribution center i to distribution center j.

**Output Format :**

The output displays "Minimum cost is: " followed by the minimum cost of the route that visits each distribution center exactly once and returns to the starting center.

**Refer to the sample output for the formatting specifications.**

**Sample testcases :**

Testcase 1 Input

Testcase 1 Output

4

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

Minimum Cost is: 80

Testcase 2 Input

Testcase 2 Output

3

0 12 8

12 0 6

8 6 0

Minimum Cost is: 26

// You are using Java

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class DeliveryRouteOptimization {

    public static int minCost = Integer.MAX\_VALUE;

    public static List<Integer> bestRoute = new ArrayList<>();

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Input number of distribution centers

        int n = sc.nextInt();

        // Input adjacency matrix

        int[][] costMatrix = new int[n][n];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                costMatrix[i][j] = sc.nextInt();

            }

        }

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

        for (int i = 0; i < n; i++) {

            currentRoute.add(i);

        }

        // Generate all permutations and find the minimum cost route

        permute(currentRoute, 0, n - 1, costMatrix);

        // Output the result

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

    }

    // Function to swap two elements in the list

    public static void swap(List<Integer> list, int i, int j) {

        int temp = list.get(i);

        list.set(i, list.get(j));

        list.set(j, temp);

    }

    // Function to generate all permutations of the list

    public static void permute(List<Integer> list, int l, int r, int[][] costMatrix) {

        if (l == r) {

            calculateCost(list, costMatrix);

        } else {

            for (int i = l; i <= r; i++) {

                swap(list, l, i);

                permute(list, l + 1, r, costMatrix);

                swap(list, l, i); // backtrack

            }

        }

    }

    // Function to calculate the cost of a specific route and update the minimum cost

    public static void calculateCost(List<Integer> route, int[][] costMatrix) {

        int currentCost = 0;

        for (int i = 0; i < route.size() - 1; i++) {

            currentCost += costMatrix[route.get(i)][route.get(i + 1)];

        }

        currentCost += costMatrix[route.get(route.size() - 1)][route.get(0)];

        if (currentCost < minCost) {

            minCost = currentCost;

            bestRoute = new ArrayList<>(route);

            bestRoute.add(route.get(0)); // return to starting point

        }

    }

}

**Problem Statement**

**You are an event coordinator tasked with organizing a tour for a group of VIP guests to visit a set of exclusive venues in a city. Given the distances between pairs of venues, your goal is to find the shortest route that visits each venue exactly once and returns to the starting venue.**

**Input Format :**

The first line of input contains an integer **n**, representing the number of venues.

The following **n** lines each contain **n** space-separated integers, representing the distances between pairs of venues.

The distance between venue i and venue j is given at the ith row and jth column of the matrix.

**Output Format :**

The first line displays "The Route is: " followed by the sequence of venues visited in the shortest path, separated by "--->".

The second line displays "Minimum cost is X", where X is the minimum cost of the shortest path found.

**Refer to the sample output for the formatting specifications.**

**Sample testcases :**

Testcase 1 Input

Testcase 1 Output

4

0 4 1 3

4 0 2 1

1 2 0 5

3 1 5 0

The Path is: 1--->3--->2--->4--->1

Minimum cost is 7

Testcase 2 Input

Testcase 2 Output

4

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

The Path is: 1--->2--->4--->3--->1

Minimum cost is 80

**Solution:**

**Refer q1 answer same**