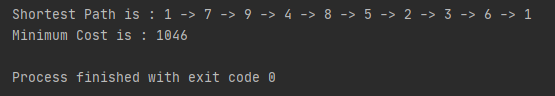
1. **Greedy Implementation**

Here we assumed as the person is starting from city 1.

Code:

import java.util.ArrayList;  
import java.util.List;  
  
public class TSPGreedy {  
 // Function to find the minimum cost path for all the paths  
 static void findMinRoute(int[][] tsp)  
 {  
 int sum = 0; // sum of the costs in min path  
 int counter = 0;  
 int j = 0, i = 0;  
 int min = Integer.*MAX\_VALUE*;  
 List<Integer> visitedRouteList = new ArrayList<>();  
 int startingNode=0;  
  
 // Starting from the 0th indexed, city 1  
 visitedRouteList.add(startingNode);  
 int[] route = new int[tsp.length];  
  
 // Traverse the adjacency matrix tsp[][]  
 while (i < tsp.length && j < tsp[i].length) {  
  
 // Corner of the Matrix  
 if (counter >= tsp[i].length - 1) {  
 break;  
 }  
  
 // If this path is unvisited then and if the cost is less then update the cost  
 if (j != i && !(visitedRouteList.contains(j))) {  
 if (tsp[i][j] < min) {  
 min = tsp[i][j];  
 route[counter] = j + 1;  
 }  
 }  
 j++;  
  
 // Check all paths from the ith indexed city  
 if (j == tsp[i].length) {  
 sum += min;  
 min = Integer.*MAX\_VALUE*;  
 visitedRouteList.add(route[counter] - 1);  
 j = 0;  
 i = route[counter] - 1;  
 counter++;  
 }  
 }  
  
 // Add the cost from the ending city to starting city  
 i = route[counter - 1] - 1;  
 sum += tsp[i][startingNode];  
  
  
 // Print path from starting node to final node  
 System.*out*.print("Shortest Path is : ");  
 for (Integer visited : visitedRouteList) {  
 System.*out*.print(visited+1+" -> ");  
 }  
  
 // Started from the node where we finished as well.  
 System.*out*.print(startingNode+1);  
  
 System.*out*.print("\nMinimum Cost is : ");  
 System.*out*.println(sum);  
 }  
  
 // Driver Code  
 public static void main(String[] args)  
 {  
 // Input Matrix  
 int[][] tsp = {  
 {-1, 225, 304, 236, 213, 339, 187, 197, 226},  
 {225, -1, 140, 153, 15, 175, 84, 160, 110},  
 {304, 140, -1, 152, 132, 41, 121, 190, 108},  
 {236, 153, 152, -1, 143, 188, 70, 73, 63},  
 {213, 15, 132, 143, -1, 166, 74, 145, 102},  
 {339, 175, 41, 188, 166, -1, 157, 226, 144},  
 {187, 84, 121, 70, 74, 157, -1, 81, 43},  
 {97, 160, 190, 73, 145, 226, 81, -1, 90},  
 {226, 110, 108, 63, 102, 144, 43, 90, -1}  
 };  
  
 // Function Call  
 *findMinRoute*(tsp);  
 }  
}

Output:



Runtime:



Conclusion

Actually, as you can see greedy approach does not giving the optimal solution. The reason is greedy algorithm consists of a sequence of decisions and each choice made should be the best possible one **at the moment.** It does not consider about whole scenario. So even though the time complexity and space complexity is better, it is this method is not good for TSP problem.

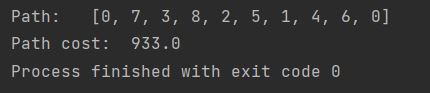
1. **Dynamic Implementation**

Here we assumed as the person is starting from city 1.

Code:

import java.util.List;  
import java.util.ArrayList;  
import java.util.Collections;  
  
public class TSP\_Dynamic {  
  
 private final int N, start;  
 private final double[][] distance;  
 private List<Integer> tour = new ArrayList<>();  
 private double minTourCost = Double.*POSITIVE\_INFINITY*;  
 private boolean ranSolver = false;  
  
 public TSP\_Dynamic(double[][] distance) {  
 this(0, distance);  
 }  
  
 public TSP\_Dynamic(int start, double[][] distance) {  
 N = distance.length;  
  
 this.start = start;  
 this.distance = distance;  
 }  
  
 // optimal tour  
 public List<Integer> getTour() {  
 if (!ranSolver) solve();  
 return tour;  
 }  
  
 // minimal tour cost  
 public double getTourCost() {  
 if (!ranSolver) solve();  
 return minTourCost;  
 }  
  
 // solutions of the traveling salesman problem  
 public void solve() {  
  
 if (ranSolver) return;  
  
 final int END\_STATE = (1 << N) - 1;  
 Double[][] memory = new Double[N][1 << N];  
  
 // Add all outgoing edges from the starting node to memory table.  
 for (int end = 0; end < N; end++) {  
 if (end == start) continue;  
 memory[end][(1 << start) | (1 << end)] = distance[start][end];  
 }  
  
 for (int r = 3; r <= N; r++) {  
 for (int subset : *combinations*(r, N)) {  
 if (*notIn*(start, subset)) continue;  
 for (int next = 0; next < N; next++) {  
 if (next == start || *notIn*(next, subset)) continue;  
 int subsetWithoutNext = subset ^ (1 << next);  
 double minDist = Double.*POSITIVE\_INFINITY*;  
 for (int end = 0; end < N; end++) {  
 if (end == start || end == next || *notIn*(end, subset)) continue;  
 double newDistance = memory[end][subsetWithoutNext] + distance[end][next];  
 if (newDistance < minDist) {  
 minDist = newDistance;  
 }  
 }  
 memory[next][subset] = minDist;  
 }  
 }  
 }  
  
 // path back to starting node and minimize cost  
 for (int i = 0; i < N; i++) {  
 if (i == start) continue;  
 double tourCost = memory[i][END\_STATE] + distance[i][start];  
 if (tourCost < minTourCost) {  
 minTourCost = tourCost;  
 }  
 }  
  
 int lastIndex = start;  
 int state = END\_STATE;  
 tour.add(start);  
  
 // get the TSP path from memory table  
 for (int i = 1; i < N; i++) {  
  
 int index = -1;  
 for (int j = 0; j < N; j++) {  
 if (j == start || *notIn*(j, state)) continue;  
 if (index == -1) index = j;  
 double prevDist = memory[index][state] + distance[index][lastIndex];  
 double newDist = memory[j][state] + distance[j][lastIndex];  
 if (newDist < prevDist) {  
 index = j;  
 }  
 }  
  
 tour.add(index);  
 state = state ^ (1 << index);  
 lastIndex = index;  
 }  
  
 tour.add(start);  
 Collections.*reverse*(tour);  
  
 ranSolver = true;  
 }  
  
 private static boolean notIn(int elem, int subset) {  
 return ((1 << elem) & subset) == 0;  
 }  
  
 public static List<Integer> combinations(int r, int n) {  
 List<Integer> subsets = new ArrayList<>();  
 *combinations*(0, 0, r, n, subsets);  
 return subsets;  
 }  
  
 private static void combinations(int set, int at, int r, int n, List<Integer> subsets) {  
  
 // Return early if there are more elements left to select than what is available.  
 int elementsLeftToPick = n - at;  
 if (elementsLeftToPick < r) return;  
  
 if (r == 0) {  
 subsets.add(set);  
 } else {  
 for (int i = at; i < n; i++) {  
 // Try to include this element  
 set |= 1 << i;  
  
 *combinations*(set, i + 1, r - 1, n, subsets);  
  
 // try with Backtrack  
 set &= ~(1 << i);  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 // Create distance weight matrix  
 int n = 9;  
 double[][] distanceMatrix = {  
 { 0, 225, 304, 236, 213, 339, 187, 197, 226 },  
 { 225, 0, 140, 153, 15, 175, 84, 160, 110 },  
 { 304, 140, 0, 152, 132, 41, 121, 190, 108 },  
 { 236, 153, 152, 0, 143, 188, 70, 73, 63 },  
 { 213, 15, 132, 143, 0, 166, 74, 145, 102 },  
 { 339, 175, 41, 188, 166, 0, 157, 226, 144 },  
 { 187, 84, 121, 70, 74, 157, 0, 81, 43 },  
 { 197, 160, 190, 73, 145, 226, 81, 0, 90 },  
 { 226, 110, 108, 63, 102, 144, 43, 90, 0 },  
 };  
  
 int startNode = 0;  
 TSP\_Dynamic solution = new TSP\_Dynamic(startNode, distanceMatrix);  
  
 System.*out*.print("Path:\t" + solution.getTour());  
 System.*out*.print("\nPath cost:\t" + solution.getTourCost());  
 }  
}

Output:



Runtime:



Conclusion

Dynamic approach gives the optimal solution for this problem. It means that provides the minimum time which required to visit all nodes (without visiting more than one time per node except the starting node). Dynamic approach considered complete scenario when giving the decried result. So, it has more time and space complexity than Greedy procedure. Due to that for a large data amount, Dynamic procedure will take more time to execute. When compare with Greedy approach, Dynamic results the reliable results.

Time complexity: O(N2 \* 2N)

Github Link: <https://github.com/Nadeesham332/TSP_PROBLEM>