Homework Assignment 5

Your assignment for Homework 5 is to write a simple recursive program to solve the Travelling Salesman Problem (TSP).

The TSP and its variants are among of the classic problems in computing. Because TSP is an NP-complete problem, we don't have a good solution for it, and we have to live with heuristics and approximations.

Finding a polynomial-time solution to the TSP would win you a million dollar prize from the Clay Institute. It's that big a problem.

You have been given a matrix of cities and pairwise distances between cities. Assume you are a salescreature based in Seattle and this is the list of cities you must visit in a cyclic tour of all the cities, with a return home to Seattle. Because distance costs money, the goal in the TSP is to find a cyclic path through the set of cities that requires the minimum total distance. (In the TSP, "distance" and "cost" are synonyms.)

You are to write a program that recursively generates all the permutations that constitute cycles through the cities, computes the distance of a complete cycle, and eventually outputs the cycle that has the minimal total distance.

The problem assumes that you are starting and ending your tour in Seattle, so there are $6! = 720$ possible cycles to look at. I recommend you trim this down for testing purposes by deleting two cities from your matrix. The resulting matrix will have $4! = 24$ possible cycles, and that's small enough for you to examine by eye.

The driver opens the input file and invokes a method to create an instance of `TSP` that will read in the input data. The driver then dumps the input data to make sure that it's been read correctly.

The TSP itself is going to be just a permutation of subscripts on the cities, so I recommend you have an `ArrayList` of cities by name and a two-dimensional `ArrayList` of `ArrayList`s of costs.

Since you not only have to compute the minimum cost, but also have to know what the cycle is that will produce the minimum cost, you will probably need an `ArrayList` for that as well.

These would be the instance variables for the `TSP` class.

You have three test files. Your code must work on all three files. Your code must also work if we change the cost values in one of these files.

THREE SUGGESTIONS:

I find that when writing programs to do recursion I need two methods and not one. My version of `solveTSP` is not the method that does recursion. Rather, it is the method that sets up the call to a method that calls itself recursively. I have seen it done otherwise, but I find I make fewer mistakes if I am more explicit in separating the setup from the actual recursion.

Second, you might well try writing the permutation-generation all by itself first. The TSP-specific part, with costs and cities, is an additional layer of complexity on top of the generation of the permutations. So first get the permutation part correct, and then add the part about looking up to find minimal costs.

Finally, think beforehand of the fact that you will have a method that is calling itself. This means that if you have to dump output for testing from this method, you will not necessarily know how many levels down in the recursion you are at any given call to the method. Recursion is especially nasty this way. You should plan either to have a "level" variable, or else make sure to dump out the partial list of permutation or remaining things to permute, so you will know where you are in the execution stack.

NOTE

It is possible to determine that a partial permutation cannot be completed to an optimal complete permutation because the partial permutation is already more costly than the current best permutation. If that's the case, then the computation can be made more efficient by an early abort strategy. You do not need to do this. That is, you can wait to make the decision on whether the full permutation is a new best solution until the complete permutation is generated.