# COMSC-200 Term Project

Instructor: Prof. Robert Burns | rburns@dvc.edu

Complete the shortest route problem that was considered in labs 5-7. This will result in a fully functional program for finding the shortest route though a network of nodes. The assignment is to find the shortest road route from **San Francisco** to **New York City**.

Note that this is NOT a solution that allows calculation of a route between any two cities in the network. That solution will come in COMSC-210 after the study of graphs and the Dykstra algorithm. The COMSC-200 solution is very specific to solving the shortest route between two preselected cities, in a network with no "dead-ends" and no "loops".

The final CPP source file is to be posted to the COMSC server. The posting is due by midnight on the day indicated on the course outline. The term project is worth 50 points.

**Solving The Shortest-Route Problem** [ `ShortestRoute.cpp` ]
In this exercise, you will build and test the third of the three classes needed to solve the shortest-route problem. Copy **ShortestRoute3.cpp** source file from **lab7**, to **ShortestRoute.cpp**.

For this assignment, write a class named **ShortestRoute**, to design and test a class that determines the shortest route between **San Francisco** to **New York City**. The route is to consist of multiple connecting `Legs`. It will include two recursive functions -- the first is simple, and just finds a valid route through the network, without regards to shortest distance. The second finds the shortest route. Here are the specifications for changing the **Route** class and for writing the **ShortestRoute** class:

**STEP 1: Write An Overloaded Stream Insertion Operator**
Convert the `outputRoute` and `outputLeg` friend functions to overloaded stream-insertion operator functions for **Route** objects. You will need this for your `main` function.

**STEP 2: Create The Network**
Add a collection (array, `vector`, etc.) of `Legs`, with at least 40 `Leg` elements. Store it as a static data member of the **ShortestRoute** class. The `Legs` must represent a network of interconnecting cities, between **San Francisco** and **New York City**. Use a map of the United States to get the names of intermediate cities, and the distance between them. Distances do *not* have to be exact, but they do have to be reasonably close, so that the results generated by your program can be verified.

Except for San Francisco, every city should have more than one path *into* it. Except for New York City, every city should have more than one path *out* of it. San Francisco is the start, and has no way into it, and New York City is the end, with no way out.

*After* your network satisfies all the above requirements, add one more leg as described here. We anticipate a **new superhighway**, with bridges, from San Francisco to New York City, going the *other way* around the world -- *west* from San Francisco, across the Pacific, across Asia and the Mediterranean, and across the Atlantic. The distance will be 21,000 miles in length, with no stops. For the *last Leg*, in your network, include this new superhighway link as a single Leg.

Carefully design your network so that there are no dead ends and no loops -- note that all legs are one-way. Accommodating these features would complicate the solution, and we handle these in Comsc-210 when we study "graphs". For purposes of our studies of OOP, what we are doing here is sufficient.

**STEP 3: Write The First Recursive Function**
Write the recursive function, `static const Route getAnyRoute(const string from, const string to)`, to create and return a **Route** object that consists of legs connecting two cities. See your lecture notes for instructions on how to write this function. To test the class, write `main` with a statement like the following:

```
cout << ShortestRoute::getAnyRoute("San Francisco", "New York City");
```

The function should throw an exception if no route can be found from the "from" city to the "to" city. If your function returns the superhighway as the solution, you did not put that Leg at the end of your database, as specified.

The function should *not* produce output -- no `cout`'s! It should return a Route object, period. Same for the 2nd function, described below.

**STEP 4: Write The Second Recursive Function**
Write the recursive function, `static const Route getShortestRoute(const string from, const string to)`, to create and return a **Route** object that consists of legs connecting two cities, such that the legs represent the shortest of all possible routes. Add to `main` with a statement like the following:

```
cout << ShortestRoute::getShortestRoute("San Francisco", "New York City");
```

The function should use STL techniques to compare and determine the shortest of among possible routes from city "A" to city "B". If no possible route exists, throw an exception. If your function returns the superhighway as the solution, you didn't do this right.

**Notes:**

- You may want to declare the **ShortestRoute** class as a friend of the other classes.
- The recursive functions call *themselves* -- they do *not* call each other.
- Use Google Maps' Get Directions option, or any competing product, to determine the distance between cities.
- Do NOT write separate H and CPP files for the classes -- this is all to be in one CPP file.

**SPECIAL SCORING RULE**
Redos are allowed just as in weekly lab assignments. But note that the point deduction for a redo is 5 instead of 2 after the last lab's due date.

Submit your completed source file to the **project** folder on the class website.

**How to work with a static vector member:**
In the class definition:
`static vector<Leg> legs;` // public or private

Below the class definition:
`vector<Leg> ShortestRoute::legs;`

In main (if `legs` is public):
`ShortestRoute::legs.push_back(Leg("City A", "City b", 123));`

In main (if `legs` is private):
`ShortestRoute::initialize();` // requires static funtion with `legs.push_back(Leg("City A", "City b", 123));` statements.

**NOTE:**
If you elect to use STL push_back, note that the function makes its own COPY of the Leg object shared via its parameter list. And Leg objects have const data members. So, you'll need to overload the assignment operator in the Leg class!

---

**How to work with a static array member:**
In the class definition:
`static const Leg legs[];` // private

Below the class definition:

```
const Leg ShortestRoute::legs[] = // database
{
  Leg("City A", "City B", 123),
  Leg("City A", "City C", 321),
  ... // other legs
};
```