Going on a Road Trip!
One of the most shared geographical visualization recently that I have noticed have been road trips! They can take on various forms:
- Shortest road trip to visit every state capital
- Optimal route to visit every national park
- Best route to visit the most interesting city in every state

Traveling Salesman Problem (TSP)
In Computer Science (and Mathematics), the Traveling Salesman Problem (TSP) asks
“What is the shortest path to visit every location exactly once?”

Consider the following graph:

![Graph Image]

Solution #1: Greedy Path Algorithm
1. Start at a random node
2. Find the edge from your current node to an unvisited node that has the minimal weight
3. Repeat Step 2 until a complete path is found

What is the shortest path using a greedy path algorithm?

Is this the shortest path?

Solution #2: Brute Force Algorithm
One of the only ways to test if our shortest path is really the shortest path is to try every single path. This can be a lot of work:

<table>
<thead>
<tr>
<th>Possible Path</th>
<th>Total Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E F G</td>
<td></td>
</tr>
<tr>
<td>A B C D E G F</td>
<td></td>
</tr>
<tr>
<td>A B C D G E F</td>
<td></td>
</tr>
<tr>
<td>A B C D G F E</td>
<td></td>
</tr>
</tbody>
</table>

How many paths are there?
In the worst case, the graph may be fully connected – every node is connected to every other node via an edge:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>
Using the NetworkX Python Graph Library

Graphs are so powerful, many libraries have been developed to make working with them in Python easier. We will use the NetworkX library, or `nx`:

```python
import networkx as nx
```

The main use of `nx` will involve the following steps:
1. Instantiate (create) a graph variable (usually called `G`)
2. Add nodes to the graph
3. Add edges to the graph
4. Perform computation on the graph

### Step 1: Instantiating the Graph

Graphs are either undirected, where edges can be traveled both ways, or directed (edges can only be traveled one way):

```python
# Create an undirected graph:
G = nx.Graph()
# Creating a directed graph:
G = nx.DiGraph()
```

### Step 2: Adding Nodes

The name of a node can be any “hashable type” in Python, which (in the context of CS 205) mostly means either a String or a Number.

```python
# Adds a node with the name "A" to graph G:
G.add_node("A")
```

### Step 3: Adding Edges

Edges are added between nodes and can, optionally, have extra attributes attached to them. In our case, we will add “weight” to the edges:

```python
# Adds an edge between "A" and "B" with weight=3
G.add_edge("A", "B", weight=3)
```

### Step 4: Performing Computation

At this point, we may need to access various properties of our graph. The common functions are:

```python
# Return a list of nodes:
G.nodes()  # ["A", "B", "C", "D", "E", "F", "G"]
# Returns a list of edges:
G.edges()  # [ ("A", "B"), ("A", "E"), ... ]
# Find if an edge exists:
G.has_edge("A", "B")  # True
# Returns a dictionary of edges for a given node:
G["B"]  # {"D": {"weight": 10}, "E": {"weight": 8},
# "A": {"weight": 3}}
# Returns the attributes about an given edge between two nodes:
G["B"]["A"]  # {"weight": 3}
```

To find a solution to TSP, we need every permutation of nodes. Python has a built-in way to find permutations of a list:

```python
from itertools import permutations
for path in permutations(G.nodes()):
    # path will contain every permutation of nodes in G
```

### Algorithm to Solve TSP:

1. 
2. 
3. 
4. 