Generating Permutations: Recursive Solution

Last time, we generated a distances matrix that contained the distances from cities in our input:

<table>
<thead>
<tr>
<th></th>
<th>New York, NY</th>
<th>Chicago, IL</th>
<th>San Francisco, CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York, NY</td>
<td>0</td>
<td>1,271,382</td>
<td>4,677,494</td>
</tr>
<tr>
<td>Chicago, IL</td>
<td>1,270,079</td>
<td>0</td>
<td>3,431,581</td>
</tr>
<tr>
<td>San Francisco, CA</td>
<td>4,675,822</td>
<td>3,429,242</td>
<td>0</td>
</tr>
</tbody>
</table>

As a reminder, we set this up as a dictionary of dictionaries so that we can access any distance with the following code:

```python
distances["New York, NY"]["Chicago, IL"]
```

The puzzle that we left with is how we generate every permutation?

To set up this problem, we will use a recursive function with two arguments:

- **path**: The current path through the graph (as a List)
- **unused**: The cities not part of the current path (as a List)

Every recursive solution almost always has three components:

1. **Recursive Case**: If there is at least one unused city, loop through all the unused cities. For each of these unused cities, make a recursive call with the city appended to the end of the path list and removed from the unused list.

Visually, we can represent the recursive step as the following tree:

```
[ NYC ] [ CHI, SFO ]
[ NYC, CHI ] [ SFO ]
[ NYC, SFO ] [ CHI ]
[ CHI, SFO ] [ ]
[ SFO ] [ CHI, NYC ]
[ ] [ NYC ] [ SFO ]
```

2. **Base Case**: When no cities remain in the unused list, return the distance and the path.

3. **Reduction**: When multiple results are returned, return the minimum of all the results.

Let's program the makePath function to complete this recursion:

```python
def makePath(path, unused, distances):
    if len(unused) == 0:
        # Base case
        return
    else:
        # Reduction result
        min = None
        # Recursive Case
        for city in unused:
            return min
```