// Charles Buyas, cjb8qf, 4-26-17, postlab11.pdf

Time Complexity:

- Prelab:
    - The time complexity of the prelab is the number of edges times log(the number of vertices).
        - This is because:
        - edges*log(edges) =
        - edges*log(vertices*vertices-1/2) =
        - edges*log((vertices^2)/2) =
        - 2*edges*log(vertices/2) =
        - edges*log(vertices)
- Inlab:
    - The time complexity of the inlab is a permutation of the number of edges.
        - This is because the worst case of a backwards-sorted vector of the locations you need to visit will end up having to run (edges!) number of times.

Space Complexity:

- Prelab:
    - The space complexity will be linear for my topological sort in the prelab.
        - This is because the number of elements I have in the map depends on how many nodes I have in the graph.
- Inlab:
    - The space complexity will be constant for my traveling.cpp in the inlab.
        - This is because each time I iterate through my while loop to check if the distance for that path is the lowest distance, I'm only utilizing the cache to run through my vector. Given that we won't have to visit more than 9 cities in any given run, we know that the final size of our vector will be about constant size.

Acceleration techniques:

- Christofides Algorithm:
    - The Christofides Algorithm uses a spanning tree to generate an Eulerian graph. With the Eulerian graph, we are able to go on an Eulerian tour. This tour cannot be more than 1.5 times the cost of the most efficient solution.
        - In order to make the graph, every vertex with an odd degree gets one matching added, effectively giving it an even degree. A spanning tree with only even numbered degrees is considered Eulerian.

        - This algorithm has a running time of Big-oh (n), which is worst case linear run time.
        - If we use this for our code, it would make run time significantly faster (compare n! to n), however it wouldn't necessarily find the best solution to our problem.
- Nearest Neighbor Algorithm:
    - This algorithm simply finds the nearest city from each city, and traverses that way. After visiting that nearest city, we mark it as visited and then find the next nearest city that hasn't been visited.
        - This could be problematic because this algorithm has a small possibility of giving you the worst possible route that you could take.
        - This has a run time of Big-theta (logV), where V is the number of vertexes.
- Branch and Bound Algorithm:
    - This algorithm recursively searches through the instances formed by a branch operation, which cuts the original graph into a set of smaller instances. After visiting each mini graph, or instance, during the recursive search is checks whether its worst case distance to travel is greater than any other instances it has already visited. If it is, it just drops it and moves on, essentially removing it from the options to travel to. It is this option removal that quickly lowers the number of searches that you need to do when working through the permutations.
        - As an example, in our inlab I wrote a while loop that checks every permutation and compares the distance variable of that permutations' vector. A way to apply the B&B (Branch and Bound) algorithm would be comparing the current distance to the already established path before where you are in the vector of strings. So say there are 10 strings to compare the distance of, and your current smallest distance is 100 units. If on the fifth string of a certain iteration you are already past 100 units of distance, say you hit 150 after the fifth distance gets added, you can immediately quit searching through that permutation to add its distances up. This eliminates tons of recursion and looping that you have to do to find the best solution.
        - Its depth, *n*, determines run time. With 2^*n* nodes, it has a worst case Big-oh (2^*n*) run time.