**StreetMap load()** is **O(N)**, where N is the number of lines in mapdata.txt. Each of these lines can be either a street name, number of segments per street, or segment of two GeoCoords. The while loop runs for N times under these conditions, assuming that each line is properly formatted (i.e. there are exactly four coordinates on a line involving GeoCoords).

**StreetMap getSegmentsThatStartWith()** is **O(1)** because this function uses a hashmap to find segments, and finding in a hashmap is performed in constant time.

**PointToPointRouter: generatePointToPointRoute()** utilizes the A\* algorithm in order to find the most efficient route in a reasonable time. The time complexity is **O(V + E)** in the worst case, where V is the number of vertices and E is the number of edges in the graph. However due to the nature of A\* having a cost function, it is statistically improbable that the path found will require testing every possible route. (Carey mentioned in office hours that A\* can be over O(N), where N is the number of segments, but we can use the prior assumption) I utilized a priority queue with a custom defined comparison operator for finding the node with the lowest f cost, and a hashmap of nodes to check if a node was in the open set.

**DeliveryOptimizer: optimizeDeliveryOrder()** utilizes the Simulated Annealing algorithm in order to find an efficient sequence to deliver items. In each iteration of the algorithm, the two-opt approach is used where two random nodes are swapped. The number of iterations in my algorithm relies on the size of the delivery vector, N.

However, calculating Big O of this is a bit complicated. My function for changing temperature was (Current Temperature – 10/(N \* log(current iteration A + 1)). The function would continually run until temperature was < 0 and iterate A each time (A starts at 1). To solve this, I manually tested the number of iterations for N from (0, 2000).

Surprisingly, the relation is approximately linear. Then to compute total distances for the old and new crow, each is O(N) respectively. Computing distances of swapped nodes relies only on the distances of adjacent nodes, making it constant as N does not affect the number of distance calculations. Therefore, the Big O of optimizer is O(3N) which simplifies to **O(N)**. I made a separate class in the support file for simulated annealing and used a vector of DeliveryRequests where two different elements were swapped each time.