Project 3 Documentation
Nicolas Rebolloso

V = # of Vertices
E = # of Edges
S = # of Students enrolled
C = # of classes per student
N = # of edges provided in toggle command

insertStudent- O(1)
Inserts using an unordered_map. This is constant time, but in the worst-case of hash collisions, it is linear to the number of students.

removeStudent- O(1)
Uses unordered_map::erase, which is constant on average but linear in the worst-case hash collision scenario.

dropClass- O(1)
Finds the student in O(1) and then iterates through a vector of size C, which has a max size of 6. This is constant.

replaceClass- O(1)
Performs map lookups and iterates through fixed-size vector of classes

removeClassFromAll- O(S)
Iterates through every student and checks their class list C, which is constant. Time is proportional to S.

toggleEdges- O(N * V)
Iterates N times for each edge pair. For each edge, it searches the adjList of a node. In a dense graph, a node's degree can be up to V.

checkEdgeStatus- O(V)
Uses the adjList for the start node, which is O(1). Iterates through its neighbors to find the target. In the worst case dense graph, a node has V - 1 neighbors.

isConnected- O(V + E)
Uses BFS to traverse the graph. In the worst case, it visits every node and every edge.

printShortestEdges- O(ElogV)
Uses Dijkstra's Algorithm with a binary heap priority queue. Each edge is processed once, and priority queue operations take log time relative to vertices. Sorting classes is O(ClogC), but C is constant.

printStudentZone- O(ElogV)
Runs Dijkstra and backtracks path parents (O(V)). Runs Prim's Algorithm on the subgraph. Since the subgraph is smaller or equal to the full graph, it is within O(ElogV)
verifySchedule- O(ElogV)
Iterates through the student's sorted schedule C - 1 times. It runs Dijkstra's algorithm to check travel times, resulting in O(ElogV).

Reflection:
This project solidified my grasp on how to work with graphs and graph algorithms. I learned how to code the actual algorithms that we learned about in class, such as Dijkstra's and Prims. One thing I would do differently if I had to start over was split up my CampusCompass logic into separate files. There are around 600 lines of code due to all the parsing, algorithm, and helper logic, and it would be best to have this split up into different files instead of different sections on the same file. I would also spend a lot more time looking at resources on how to implement the needed algorithms to get an idea of what to do instead of first trying to implement them by myself, which took a lot of time away.

Coming into the project I was intimidated because graphs are a pretty abstract concept that seem challenging to code. I began with coding the parsing and validation logic which was frustrating but conceptually simple. Once that was done, coding the graphing algorithms seemed like a huge step up, but in reality it wasn't so bad. Taking each algorithm step-by-step, line-by-line helped me build a strong understanding of how these implementations work. I believe this project has helped me build a much stronger and deeper understanding of not only graphs but also some of the other data structures used in their implementations.