Design thinking process: For this project, my aim was to keep the process as simple and efficient as possible. With this in mind, I decided to make use of the algs4.jar file from the Princeton Algorithms book website (1), from which I could export all needed classes from the book. By using this jar file as an external library, I could fit all of the code needed into one class. This would avoid having to create several Java classes, keeping the code more straightforward to write and read.

Part 1: Since the input files contained so many data entries, a data structure with a low speed complexity would be needed so that the data could be accessed as quickly as possible. With this in mind, I decided to use hash maps to store the data from each file (stops.txt, stop_times.txt and transfers.txt). They have a worst case speed complexity of O(n), and a best case complexity of O(1). This was very important as the data retrieval time was vital to ensure that the program ran smoothly and effectively, with respect to the amount of data. For this part, I used a hashmap to add data from the stops.txt file, for the bus stop numbers.
I chose to use a buffered reader to read in the data from the stops.txt file. I did this so that I could skip the first line of the line easily, as the first line contains labels of the data alone, which are not needed. The buffered reader also allowed me to split up data entries by the "," character, which made organising the data into the hashmaps much easier.
From the algs4 jar file, I implemented the EdgeWeightedDigraph, DirectedEdge and DijkstraSP classes. I used the Array List data structure, with the DirectedEdge class as the object, in order to store the edges and their associated costs from transfers.txt and stop_times.txt. The time complexity for getting an element by index in Array Lists is O(1), which is why I decided to implement this data structure. I used the EdgeWeightedDigraph class with the number of bus stops from stops.txt to create an empty edge-weighted digraph needed for the DijkstraSP class.
After getting the first and second bus stop number from the user, I used DijkstraSP to create a DijkstraSP object with the EdgeWeightedDigraph object I previously created and the first stop number. I then used this DijkstraSP object with the distTo() function to find the shortest path. Finally, I printed out my ArrayList with all edges and costs enroute.

Part 2: I used a hashmap again to store the data needed for part 2. The stops.txt input file contained all names of the bus stops, which was needed for the search functionality. I chose to use the TST (Ternary Search Tree) class from the Algorithms algs4 jar file, and

its time complexity is proportional to the height of the search tree, and its space is proportional to the length of the string to be stored.
I used a buffered reader in this part also to separate the bus stop names in the input file, and added them to the empty TST for effective storage.
When I got the partial or full bus stop name from the user, I called the TST class to make a TST object, with the entered bus stop name as the key. I then printed the user’s bus stop name as an iterable string with the TST call, as TST requires an iterable string as input, and then proceeded to print out all related bus stop names.

Part 3: This part was very similar to the previous searching utility, except the arrival times from the stop_times file were used to find the corresponding trip id. I used the buffered reader again, to read in the file. I also reused a hashmap in the same way as part 2. The arrival time was inputted by the user, and this time was used as a string in the TST call, as the key for the trip id.

References:

(1) Algorithms Fourth Edition by Robert Sedgewick and Kevin Wayne. Princeton website for algs4.jar file: https://algs4.cs.princeton.edu/code/.