Pseudocode: RouteFinder

1. **Package and Imports:**
    - Define the package as com.example.
    - Import necessary libraries:
        - Standard libraries for networking, collections, I/O, etc.
        - JSON handling libraries (org.json.JSONArray, org.json.JSONObject).
        - External libraries for handling environment variables (e.g., dotenv).
2. **Class: RouteFinder:**
    - **Private Static Variable:**
        - mapboxApiKey: Used to store the Mapbox API key loaded from environment variables.
    - **Static Block:**
        - Load environment variables using Dotenv.load() and retrieve the Mapbox API key.
3. **Method: main(String[] args)**
    - **Purpose:** Entry point of the program.
    - **Parameters:** Accept user input (e.g., starting and destination locations).
    - **Steps:**
        - Parse the input arguments.
        - Call a helper function to construct the request URL for Mapbox API.
        - Send the HTTP request and get the response.
        - Parse the JSON response to extract relevant route information.
        - Print or display the route details (e.g., distance, duration).
4. **Method: getRoute(String start, String destination)**
    - **Purpose:** Retrieve route information from Mapbox API.
    - **Parameters:**
        - start: The starting location.
        - destination: The destination location.
    - **Steps:**
        - Build the URL for the Mapbox Directions API using the start and destination locations.

        - Open an HTTP connection to the URL.
        - Handle errors in case of invalid responses (e.g., non-200 status codes).
        - Read the response using BufferedReader.
        - Convert the response into a JSON object.
        - Extract relevant details such as distance, duration, and route coordinates.
        - Return the parsed route information.
5. **Method: buildURL(String start, String destination)**
    - **Purpose:** Construct the URL to query Mapbox Directions API.
    - **Parameters:**
        - start: Starting location in latitude and longitude format.
        - destination: Destination location in latitude and longitude format.
    - **Steps:**
        - Append start and destination coordinates to the base Mapbox API URL.
        - Include necessary parameters (e.g., access token).
        - Return the constructed URL as a string.
6. **Method: parseRouteData(JSONObject jsonResponse)**
    - **Purpose:** Parse the JSON response from the Mapbox API.
    - **Parameters:**
        - jsonResponse: The raw JSON response from the API.
    - **Steps:**
        - Extract the routes array from the JSON.
        - For the first route, extract:
            - distance: Total distance of the route.
            - duration: Estimated travel time.
            - geometry: Route path as a series of coordinates.
        - Store the parsed data in an appropriate format (e.g., custom object or dictionary).
        - Return the extracted route information.
7. **Helper Methods:**
    - **Helper Method: sendHTTPRequest(String url)**

        - Open a connection to the provided URL.
        - Handle potential exceptions such as IOException.
        - Return the response as a string.
    - **Helper Method: readResponse(BufferedReader reader)**
        - Read the lines from the input stream.
        - Concatenate the lines to form the complete response.
        - Return the response as a single string.
8. **Error Handling:**
    - Use try-catch blocks around network operations (e.g., opening connections, reading responses).
    - Ensure proper resource management (e.g., closing connections and streams in finally block).
    - Log errors and notify the user of issues such as network failures or invalid locations.

Algorithm Implemented: Dijkstra's Algorithm

1. **Initialize:**
    - Create a priority queue.
    - Set the distance to the starting node to 0 and all other nodes to infinity.
    - Set the previous node for each node to null.
2. **While the priority queue is not empty:** a. Extract the node with the smallest distance (current node). b. If the current node is the destination, construct the path:
    - Initialize an empty list for the path.
    - Traverse backwards from the destination to the start using the previous node references.
    - Reverse the path list to get the correct order.
    - Return the path. c. If the smallest distance is infinity, break (all remaining nodes are unreachable).
3. **For each neighbor of the current node:** a. Calculate the tentative distance from the start node to the neighbor. b. If the tentative distance is less than the currently known distance:
    - Update the neighbor's distance.
    - Update the previous node reference for the neighbor.

    - Remove the neighbor from the priority queue.
    - Add the neighbor back to the priority queue with the updated distance.
4. **If the destination is unreachable, return an empty path.**
5. **Return the constructed shortest path.**