Algorithm 2: Greedy Approach to Hamiltonian Problem

a. Ryan Monte – ryanmonte@csu.fullerton.edu
Angel Penaloza – angelpenlza@csu.fullerton.edu
Carlos Lopez – carlosjlopezz@csu.fullerton.edu
Boushra Bettir – boushra.bettir04@csu.fullerton.edu

b. *Pseudocode*:

Problem: Find the preferred starting city such that you can complete a full circle around all cities, starting and ending at the same city, without running out of fuel.

Input:

- `city_distances`: An array of integers representing the distance between neighboring cities.
- `fuel`: An array of integers representing the amount of fuel available at each city's gas station.
- `mpg`: An integer representing the number of miles the car can drive per gallon of fuel.

Output:

- The index of the preferred starting city.

Constraints and Assumptions:

- There are valid integers entered for city distances, fuel, and mpg. The integers entered for these variables must make it possible to solve the problem. For example, the mpg cannot be a negative number

---

```
FUNCTION findPreferredStartingCity(city_distances, fuel, mpg):
   # Step 1: Initialize necessary variables
   n = LENGTH(city_distances)  # Number of cities

   total_fuel = 0              # Total fuel gained across the journey
   total_distance = 0          # Total distance that needs to be covered
   current_fuel = 0            # Fuel balance as we travel from city to city
   start_city = 0              # The city from where we start our journey

   # Step 2: Traverse through all the cities
   FOR i FROM 0 TO n-1:
      # Calculate fuel gained in this city and fuel required to travel to the next city
      fuel_gained = fuel[i] * mpg
      distance_to_next = city_distances[i]
```

```
        # Update total fuel and distance counters
        total_fuel += fuel_gained
        total_distance += distance_to_next

        # Update current fuel after moving to the next city
        current_fuel += fuel_gained - distance_to_next

        # Step 3: If at any point the current fuel is less than 0, we can't complete the
    journey from this start
        IF current_fuel < 0:
           # Reset the starting city to the next city
           start_city = i + 1

           # Reset the current fuel balance (as we are starting over)
           current_fuel = 0

     # Step 4: Check if total fuel is enough to cover the total distance (guaranteed
    by problem)
     IF total_fuel >= total_distance:
        RETURN start_city
     ELSE:
  RETURN -1  # Edge case if no valid solution exists, though the problem
guarantees one
```

c. *Mathematical analysis and Big O Efficiency of the Pseudocode:*

To analyze the Big O efficiency class of the pseudocode provided for the Hamiltonian problem using a step count method, we will examine each component of the pseudocode and determine the number of operations performed based on the input size n, which is the number of cities.

Step-by-Step Analysis:

1. **Initialization**:
   - `n = LENGTH(city_distances)` takes $O(1)$ time.
   - Initializing `total_fuel`, `total_distance`, `current_fuel`, and `start_city` involves a constant number of assignments, also taking $O(1)$ time.

2. **Loop through all cities** (from 0 to $n-1$):
   - The for-loop iterates n times, where n is the number of cities.
   - In each iteration:
     - Calculating `fuel_gained = fuel[i] * mpg` takes $O(1)$ time.
     - Accessing `city_distances[i]` for `distance_to_next` takes $O(1)$ time.

- Updating `total_fuel` and `total_distance` takes ***O*(1)** time.
- Updating `current_fuel` takes ***O*(1)** time.
- The `IF current_fuel < 0` condition check and resetting `start_city` (when necessary) also take ***O*(1)** time.

Since all the operations inside the loop are constant time operations, the loop runs in ***O*(*n*)**.

3. **Final Check**:
   - After the loop, we perform a constant-time check `IF total_fuel >= total_distance` which takes ***O*(1)** time.

Total Time Complexity:

- The initialization phase takes ***O*(1)** time.
- The loop runs n times, with each iteration involving constant-time operations, so it takes ***O*(*n*)** time.
- The final check takes ***O*(1)** time.
-

Thus, the total time complexity of the algorithm is:

$O(1) + O(n) + O(1) = O(n)$

The algorithm runs in linear time, ***O(n)*** where n is the number of cities.

Proof by Step Count:

Each step in the algorithm either takes constant time (***O(1)***) or is part of a loop that iterates ***n*** times. Since no nested loops or operations with higher complexity exist, the algorithm's performance grows linearly with respect to the number of cities.