Group 1:

Quan Truong (hungquantruong@csu.fullerton.edu)

Andrew Conti (andrewconti@csu.fullerton.edu)

Arai Leyva (araileyva@csu.fullerton.edu)

Eliot Park (EliotNah@csu.fullerton.edu)

Project 1

CPSC 335 Spring 2025

Algorithm 1: Greedy Approach to Hamiltonian Problem

Understanding the problem: Given the number of cities i and gallons of fuel, we can refill at each city i, respectively. We need to find a city to start from, and as long as we continue the journey(visiting and refueling at each city), we will have enough fuel to visit all the cities.

Mathematical analysis:
At the start of the problem, we were given an array for distance between the cities where distance[i] is the distance between city i and city i+1. We are also given an array of fuel each city can have; for example, city 1 has 3 gallons we can refill, so fuel[1] = 3.
We were also given the car's mpg, which is 10.
The fuel tank is always empty initially, and it must always be a positive number to travel through all the cities.

- At each city i, you can fill up with fuel[i] gallons.
- Since the car can travel mpg = 10 miles per gallon, the effective miles you can drive from city i after refueling is:

max miles from city i = fuel[i] * mpg

Total_fuel = fuel[0]* mpg - distance[0] + … + fuel[i] * mpg -distance[i]

Pseudocode:

```
Algorithm FindPerferredStartingCity(distances[], fuel[], mpg):
    num_cities ← LENGTH(distances)
    total_fuel_surplus ← 0
    current_fuel ← 0
    start_city ← 0

    For i from 0 to numCities - 1 do
        net_fuel ← fuel[i] * mpg - distances[i]
        total_fuel_surplus ← total_fuel_surplus + net_fuel
        current_fuel ← current_fuel + net_fuel

        If current_fuel < 0 then
            start_city ← i + 1
            current_fuel ← 0
```

```
    End if
End
```

Analysis:

- Time Complexity is O(n) because the algorithm only passes a single time through the list of cities.
- Space complexity is O(1): The algorithm does not allocate any additional space due to using a fixed number of variables, and it does not scale therefore, it remains constant.

Algorithm 2: Connecting Pairs of Persons

Pseudocode:

```
Algorithm getMinimumSwaps(row[])
        Swaps ← 0
        For i = 0, i = row length, i +2 do
                If math.floor(row[i]/2) != math.floor(row[i]/2) then
                        For j = i+1, row length do
                                If math.floor(row[j]/2) = math.floor(row[i]/2) then
                                        Swap(row[i+1], row[j])
                                        Swaps++
                                End If
                        End For
                End If
        End For
        Return Swaps
```

Mathematical analysis:

Time Complexity is O(n^2) because there is an inner loop that is nested with the outer loop. The outer loop iterates over the list while incrementing the index by 2 in each iteration, so it runs approximately n times where n is the length of the list. The nested iteration shows the time complexity of O(n) * O(n) = O(n^2). It ends up being this complex because, for each element in the outer loop, the function may have to check every element to find the correct pair, which can increase the number of operations.

Best Case: When the list is arranged where the couples are all sitting next to each other, the time complexity will be O(n) due to the function only needing to verify that it is correctly seated without performing any swaps.

Worse Case: If the couples are entirely mismatched, the maximum number of swaps is required for the correct arrangement. The inner loop may need to iterate over the remaining elements to find the proper match for every pair in the outer loop. This causes the complexity to be O(n^2)because the algorithm needs to perform a swap for each element required to arrange all the couples correctly.

Space complexity is O(1), and the algorithm uses a constant amount of extra space with variables such as minimum and temp. It essentially modifies the list in place and doesn't use any other data structures that allow the input size to grow therefore, it is O(1)**.**