Project 1

About	
Name	Nathan
Email	nathan-chen@csu.fullerton.edu

Algorithm 1: Connecting Pairs of Persons

Proving by induction

- For an array with length *n*, the worst case scenario occurs when the array is in reverse order: `row = [n, n-1, ..., 2, 1]`.
    - In this case every element must be swapped to its correct position.
- **Base Case**: for `n=2`, `row` can either be `[1, 2]` or `[2, 1]`.
    - In both cases, only one swap is needed.
    - Base case holds.
- **Inductive Step**: Assume that for an array length *k*, the number of swaps required is `k / 2`.
    - In an array with length `k + 2`, where the first k elements are in reverse order, and the last two elements are in the correct positions
        - `[k + 1, k, ..., 2, 1, k + 2, k + 3]`
    - In order to place `k + 1` in its correct position, it needs to be swapped with k, which requires 1 swap.
        - After swapping: `[k, k + 1, ..., 2, 1, k + 2, k + 3]; the remaining elements are in reverse order, forming an array of length *k*.
    - By the inductive hypothesis, the remaining swaps fequired for the array length *k* is `k / 2`
    - Therefore, the total number of swaps required for an array of length `k + 2` is 1 (for swapping *k* and `k + 1`) + `k / 2` (for the remaining swaps).
        - `(k + 2) / 2`
- **Conclusion**: By induction, we can conclude for that an array of length *n*, the algorithm has a time complexity of `O(n/2) = O(n)`.
    - The time spent grows linearly with the size of the array.

Pseudocode

```
Function min_swaps(row[]) {
    int swaps = 0
    for each couple (i, i + 1) from 0 to row.length() with 2 step increments {
        if floor(row[i] / 2) != floor(row[i + 1] / 2) {
            find j where floor(row[j] / 2) == floor(row[i] / 2)
```

```
            swap row[i + 1] with row[j]
            swaps++
        }
    }
    return swaps
}
```

How to run:

```
~$ python3 algorithm-1.py
```

Algorithm 2: Greedy Approach to Hamilton Problem

Proving time complexity through induction

- Let the number of cities be *n*
- **Base Case**: For `n = 1`, the function returns zero.
    - Time complexity for `n = 1` is O(1)
- **Inductive step**: Assuming that for `n = k`, the function has a time complexity of `O(k)`.
    - For `n = k + 1`: in the worst case, the car starts at city *0* and cannot reach city `k + 1`.
        - The function will iterate through all cities from *0* to *k* before determining that city *0* is not a valid starting point.
    - During each iteration, the function calculates the total gas and total distance, which are `O(1)` operations.
        - Therefore, the time complexity of the function for `n = k + 1` is `O(k + 1) = O(k)`.
    - **Conclusion**: By induction, the function has a time complexity of O(n) for an input array of size *n*.
        - The time taken grows linearly with the number of cities

Pseudocode

```
Function find_starting_city(city_distances, fuel, mpg):
    total_gas = 0
    total_distance = 0
    start_city = 0

    For i from 0 to len(city_distances) - 1:
        total_gas += fuel[i]
        total_distance += city_distances[i]

        If total_gas * mpg < total_distance:
            start_city = i + 1
            total_gas = 0
            total_distance = 0
```

```
    Return start_city % len(city_distances)
```

How to run:

```
~$ python3 algorithm-2.py
```