Azaan Mavandadipur (azaanmp2@csu.fullerton.edu)
Kevin Baek (kbee@csu.fullerton.edu)
Omar Montes (621omontes@csu.fullerton.edu)

Project 1

Algorithm 1

Pseudocode:

Question: How to connect pairs of persons
Answer: Minimize Swaps to seat couples side by side

Problem: Given a row of seats where 2n individuals are sitting, ensure all couples (in these pairs (0,1), (2,3) …, (2n-2, 2n-1)) are sitting side by side with the minimum number of swaps
Input: A list row[] of length 2n where each entry is an integer representing a person.
Output: The minimum number of swaps required so that every couple is seated next to each other.

Pseudocode (C++)

```
Function minSwapsToSeatCouples(row)
        swaps = 0
        for i from 0 to length of row - 1, step 2
                partner = findPartner(row[i])

                if row[i + 1] != partner
                        partnerIndex = findPartnerIndex(row, partner)
                        swap(row[i + 1], row[partnerIndex]
        return swaps

function findPartner(person)
```

```
    if person is even
        return person + 1
    else
        return person - 1

Function findPartnerIndex(row, partner)
    for j from 0 to length of row - 1
        If row[j] == partner
            return j
```

Efficiency Class using Step Counts

function minSwapsToSeatCouples

- Outer loop for i from 0 to length of row - 1, step 2 is O(n/2) as it iterates through the pairs in array.
- partner = findPartner(row[i]) is called O(1) conditional check + O(1) for arithmetic
- if row[i + 1] != partner is O(1) for the comparison
- partnerIndex = findPartnerIndex(row, partner) is called which iterates through an array for j from 0 to length of row - 1, worst case being n, O(n)
- swapping partner takes 3 steps each O(1) + O(1) + O(1)

Adding all the steps we get n/2 * (n +6) = (n^2)/2 +3n. n^2 is the highest so complexity is O(n^2)

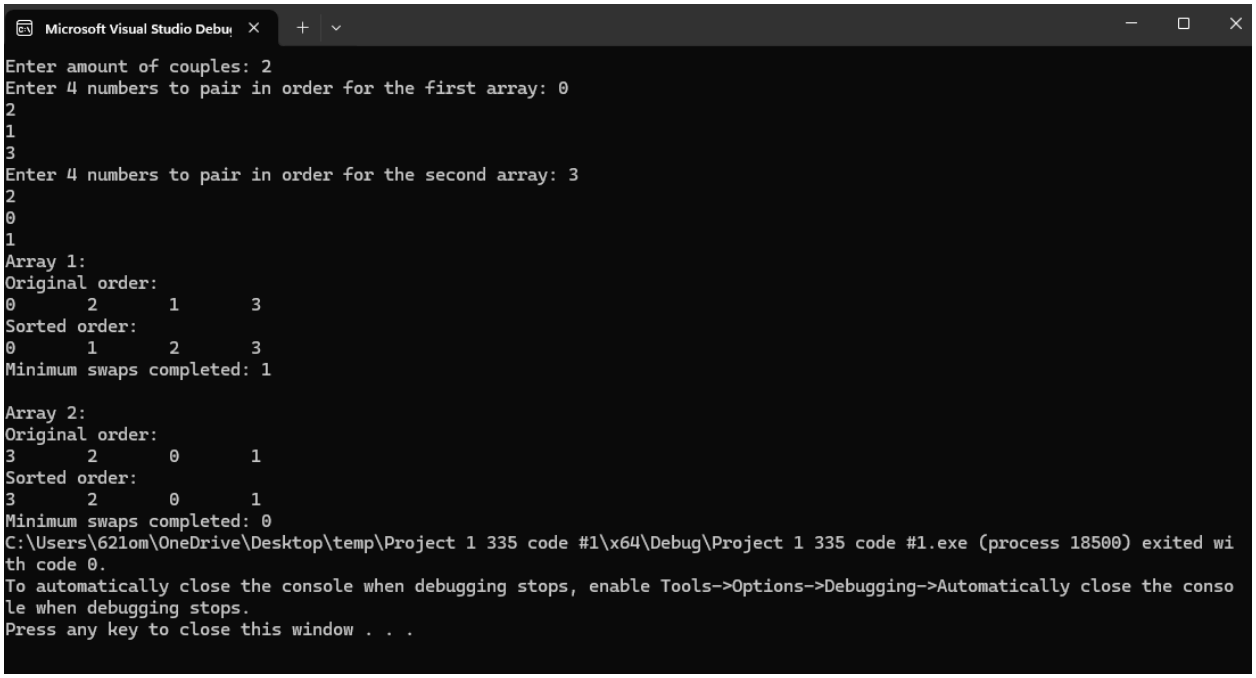
Code for Algorithm 1 (C++)

```
#include <iostream>
#include <vector>

//function to see if partner is next to person
```

```
int findPartner(int person) {
	if (person % 2 == 0) {
		return person + 1;
	}
	else {
		return person - 1;
	}
};

//function to find partner index
int findPartnerIndex(std::vector<int>& row, int partner) {
	for (int i = 0; i < row.size(); i++) {
		if (row[i] == partner) {
			return i;
		}
	}
	return -1;
};

//function to find how many swaps it takes to get couples together
int minSwapsToSeatCouples(std::vector<int>& row) {
	int swaps = 0;


	for (int i = 0; i < row.size() - 1; i += 2) {
		int partner = findPartner(row[i]);

		if (row[i + 1] != partner) {
			int partnerIndex = findPartnerIndex(row, partner);
			int temp = row[i + 1];
			row[i + 1] = row[partnerIndex];
```

```
                        row[partnerIndex] = temp;

                        swaps++;
                }
        }
        return swaps;
};


int main() {
        int nBy2 = 0;

//user input for number of couples
        std::cout << "Enter amount of couples: ";
        std::cin >> nBy2;
        int n = nBy2*2;
        std::vector <int> row1(n);
        std::vector <int> row2(n);
//user input for what number each index will be
        std::cout << "Enter 4 numbers to pair in order for the first array: ";
        for (int i = 0; i < 4; i++) {
                std::cin >> row1[i];
                }
//second user input for what number each index will be
        std::cout << "Enter 4 numbers to pair in order for the second array: ";
        for (int i = 0; i < 4; i++) {
                std::cin >> row2[i];
        }

        std::cout << "Array 1: \n";
        std::cout << "Original order: " << "\n";
```

```
    for (int i = 0; i < n; i++) {
        //for loop to print array
        std::cout << row1[i] << "\t";
    }
    int minSwaps1 = minSwapsToSeatCouples(row1);
    std::cout << "\n" << "Sorted order: " << "\n";

    for (int i = 0; i < n; i++) {
        std::cout << row1[i] << "\t";
    }
    std::cout << "\n" << "Minimum swaps completed: " << minSwaps1 << "\n\n";

    std::cout << "Array 2: \n";
    std::cout << "Original order: " << "\n";
    for (int i = 0; i < n; i++) {
        std::cout << row2[i] << "\t";
    }
    int minSwaps2 = minSwapsToSeatCouples(row2);
    std::cout << "\n" << "Sorted order: " << "\n";
    for (int i = 0; i < n; i++) {
        std::cout << row2[i] << "\t";
    }
    std::cout << "\n" << "Minimum swaps completed: " << minSwaps2;

}
```



Algorithm 2

Pseudocode:

Question: How do you return to the start with 0 or more gallons of gas?

Answer: Write a function that returns the index of the preferred starting city.

Problem: Find the index of the preferred starting city such that a car can complete a circular trip, refueling at each city's gas station, and return to the original city with 0 or more gallons of fuel left.

Input:

- city_distances[]: An array representing the distances between the cities in a circular road
- fuel[]: An array representing the amount of fuel available at each city
- mpg: An integer representing the number of miles per gallon traveled.

Output: The index of the preferred starting city.

Pseudocode (C++)

Function findPreferredStartingCity(city_distances, fuel, mpg)

```
n = length of city_distances
total_fuel = 0
current_fuel = 0
Starting_city = 0

for i from 0 to n - 1
        fuel_gained = fuel[i] * mpg

        total_fuel += fuel_gained - city_distances[i]
        current_fuel += fuel_gained - city_distances[i]

        if current_fuel < 0
                starting_city = i + 1
                current_fuel = 0

if total_fuel >= 0
        return starting_city
else
        return -1
```

Efficiency Class using Step Count

- Function findPreferredStartingCity(city_distances, fuel, mpg)
- loop for i from 0 to n - 1 loop count iteration will be n.
- fuel_gained, total_fuel, current_fuel each take 1 operation each for the arithmetic
- if current_fuel < 0 takes 1 operation for conditional check
- if total_fuel >= 0 takes 1 operation for conditional check.

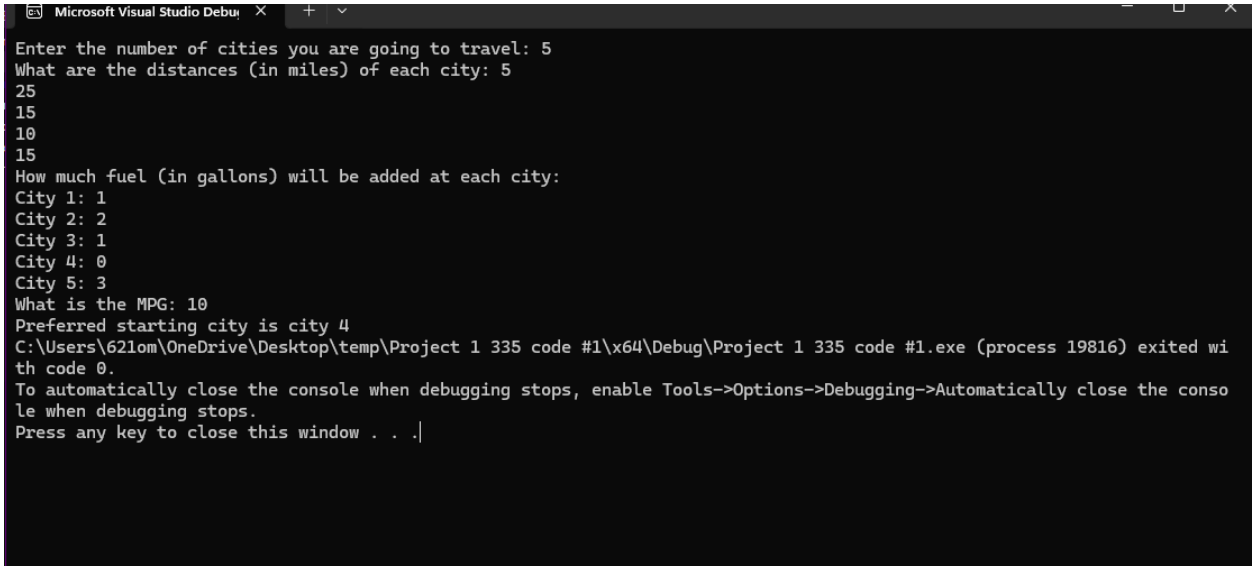
Total time operations is n * (1 + 1 + 1 + 1 +1 + 1) = 6n

Efficiency class using step count is O(n)

Code for Algorithm 2 (C++)

```
#include <iostream>
#include <vector>

//FindPCS = preferred starting city function
//Returns index of preferred city
int findPSC(std::vector<int>& cityDistances, std::vector<int>& fuel, int mpg) {
	int totalFuel = 0;
	int currentFuel = 0;
	int startingCity = 0;

	for (int i = 0; i < cityDistances.size(); i++) {
		int fuelGained = fuel[i] * mpg;
		totalFuel += fuelGained - cityDistances[i];
		currentFuel += fuelGained - cityDistances[i];

		if (currentFuel < 0) {
			startingCity = i + 1;
			currentFuel = 0;
		}
	}
	if (totalFuel >= 0) {
		return startingCity +1;
	}
	else {
		return -1;
	}
};

int main() {
```

```
        int numCities = 0;
//input variables for function
        std::cout << "Enter the number of cities you are going to travel: ";
        std::cin >> numCities;
//vectors for easy input of elements
        std::vector<int> cityDistances(numCities);
        std::vector<int> fuel(numCities);
        int mpg = 0;
        std::cout << "What are the distances (in miles) of each city: ";
        for (int i = 0; i < numCities; i++) {
                std::cin >> cityDistances[i];
        }
        std::cout << "How much fuel (in gallons) will be added at each city: \n";
        for (int i = 0; i < numCities; i++) {
                std::cout << "City " << i+1 << ": ";
                std::cin >> fuel[i];
        }
        std::cout << "What is the MPG: ";
        std::cin >> mpg;
// finding which city is easiest to start with
        int startingCity = findPSC(cityDistances, fuel, mpg);

        std::cout << "Preferred starting city is city " << startingCity - 1;
}
```



Algorithm 3

Pseudocode

Question:
Answer

Problem: Given the schedules of two or more group members, their daily active periods and the duration of a meeting, find the available time intervals when all members are free to meet for at least the specified duration

Input:

- Busy_Schedules[]: A 2D list where each sub-array represents the time intervals that a group member is available
- Working_periods[]: An array that contains each member's daily working period, representing the earliest time they are available and the latest time they are available

- Duration_of_meeting[]: The minimum duration (in minutes) of the desired meeting

Output: A list of available time intervals when all group members can meet for at least the given duration.

Pseudocode (C++):

```
function findAvailableSlots(Busy_Schedules, Working_periods, duration_of_meeting):
    n = number of members
    available_slots = []

    # Step 1: Convert time from 'HH:MM' to minutes for easier calculations
    for each member in Busy_Schedules and Working_periods:
        convert all times in Busy_Schedules to minutes
        convert Working_periods to minutes

    # Step 2: Merge busy times of all members
    merged_busy_intervals = mergeBusyIntervals(Busy_Schedules)

    # Step 3: Find the available time slots for all members
    for each member in Working_periods:
        earliest = member's earliest working period
        latest = member's latest working period

        # Available times are between the end of one busy interval and the start of the next
        for each interval in merged_busy_intervals:
            if the interval starts after earliest:
                available_slot = (earliest, interval_start)
                if the duration of available_slot is at least duration_of_meeting:
                    add available_slot to available_slots
                earliest = interval_end
```

```
        # Add any remaining time after the last busy period
        if earliest < latest:
            available_slot = (earliest, latest)
            if the duration of available_slot is at least duration_of_meeting:
                add available_slot to available_slots

    # Step 4: Convert times back to 'HH:MM' format for output
    for each slot in available_slots:
        convert time from minutes back to 'HH:MM'

    # Step 5: Return the available slots sorted in ascending order
    return available_slots



function mergeBusyIntervals(Busy_Schedules):
    merged_intervals = []

    # Flatten all busy schedules into one list of intervals
    all_intervals = flatten(Busy_Schedules)

    # Sort intervals by start time
    sort all_intervals by start time

    # Merge overlapping or adjacent intervals
    for each interval in all_intervals:
        if merged_intervals is empty or current interval does not overlap with the last merged
interval:
            add current interval to merged_intervals
        else:
            # Merge the current interval with the last interval
```

merged_intervals[last] = merge last interval and current interval

return merged_intervals

Efficiency Class using Step Count

Step 1: Converting Busy_Schedules and Working_periods from HH to minutes for each member takes time of n depending on how many members there are. O(n)

Step 2: merged_busy_intervals = mergeBusyIntervals(Busy_Schedules) total number of busy intervals for all members will be n. Sorting and merging will be considered n log n. O(N log N)

Step 3: for each interval in merged_busy_intervals N need to iterate through all the merged busy intervals which is checked against each members intervals N. O(n^2)

Step 4: Function to convert times back to HH will take N times depending on how many members there are. O(n)

Step 5: Returning the available time slots proportional to the number of members will be O (N log N)

O(n) + O(N log N) + O(n^2) + O(n) + O(N log N)

Time complexity O(n^2).

Code:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
```

```
using namespace std;

// Function to convert time in "HH:MM" format to minutes
int timeToMinutes(const string& time) {
    int hours = stoi(time.substr(0, 2));
    int minutes = stoi(time.substr(3, 2));
    return hours * 60 + minutes;
}

// Function to convert minutes to "HH:MM" format
string minutesToTime(int minutes) {
    int hours = minutes / 60;
    minutes = minutes % 60;
    string hourStr = (hours < 10) ? "0" + to_string(hours) : to_string(hours);
    string minuteStr = (minutes < 10) ? "0" + to_string(minutes) : to_string(minutes);
    return hourStr + ":" + minuteStr;
}

// Function to merge busy intervals
vector<pair<int, int>> mergeBusyIntervals(vector<vector<pair<int, int>>> busySchedules) {
    vector<pair<int, int>> merged;

    // Add all busy intervals into one vector
    for (const auto& schedule : busySchedules) {
        for (const auto& interval : schedule) {
            merged.push_back(interval);
        }
    }

    // Sort intervals by start time
    sort(merged.begin(), merged.end());
```

```
    // Merge overlapping intervals
    vector<pair<int, int>> result;
    result.push_back(merged[0]);

    for (int i = 1; i < merged.size(); i++) {
        if (merged[i].first <= result.back().second) {
            // If intervals overlap, merge them
            result.back().second = max(result.back().second, merged[i].second);
        } else {
            result.push_back(merged[i]);
        }
    }

    return result;
}

// Function to find available slots between merged busy intervals
vector<pair<int, int>> findFreeIntervals(vector<pair<int, int>> mergedBusyIntervals, int
dailyStart, int dailyEnd, int duration) {
    vector<pair<int, int>> freeIntervals;
    int previousEnd = dailyStart;

    for (const auto& interval : mergedBusyIntervals) {
        int start = interval.first;
        if (start - previousEnd >= duration) {
            freeIntervals.push_back({previousEnd, start});
        }
        previousEnd = max(previousEnd, interval.second);
    }
```

```
    // Check for available time after the last busy interval until the end of the working period
    if (dailyEnd - previousEnd >= duration) {
        freeIntervals.push_back({previousEnd, dailyEnd});
    }

    return freeIntervals;
}

// Main function to find available meeting slots
vector<pair<string, string>> findAvailableSlots(vector<vector<pair<string, string>>>
busySchedules, vector<pair<string, string>> workingPeriods, int duration) {
    int n = busySchedules.size();

    // Convert all times in busySchedules and workingPeriods to minutes
    vector<vector<pair<int, int>>> busySchedulesInMinutes(n);
    for (int i = 0; i < n; i++) {
        for (const auto& interval : busySchedules[i]) {
            busySchedulesInMinutes[i].push_back({timeToMinutes(interval.first),
timeToMinutes(interval.second)});
        }
    }

    // Merge busy intervals from all members
    vector<pair<int, int>> mergedBusyIntervals = mergeBusyIntervals(busySchedulesInMinutes);

    // Determine the global working period (intersection of all members' working periods)
    int globalStart = timeToMinutes(workingPeriods[0].first);
    int globalEnd = timeToMinutes(workingPeriods[0].second);
    for (int i = 1; i < n; i++) {
        globalStart = max(globalStart, timeToMinutes(workingPeriods[i].first));
        globalEnd = min(globalEnd, timeToMinutes(workingPeriods[i].second));
```

```
    }

    // Find free intervals within the global working period
    vector<pair<int, int>> freeIntervals = findFreeIntervals(mergedBusyIntervals, globalStart,
globalEnd, duration);

    // Convert free intervals back to "HH:MM" format
    vector<pair<string, string>> result;
    for (const auto& interval : freeIntervals) {
        result.push_back({minutesToTime(interval.first), minutesToTime(interval.second)});
    }

    return result;
}

int main() {
    // Sample input
    vector<vector<pair<string, string>>> busySchedules = {
        {{"07:00", "08:30"}, {"12:00", "13:00"}, {"16:00", "18:00"}},  // person 1 schedule
        {{"09:00", "10:30"}, {"12:20", "13:30"}, {"14:00", "15:00"}, {"16:00", "17:00"}}  //
person 2 schedule
    };

    vector<pair<string, string>> workingPeriods = {
        {"09:00", "19:00"},  // person 1 working period
        {"09:00", "18:30"}   // person 2 working period
    };

    int duration = 30;  // Meeting duration in minutes

    // Find available slots
```

```
    vector<pair<string, string>> availableSlots = findAvailableSlots(busySchedules,
workingPeriods, duration);

    // Output available meeting times
    cout << "Available meeting slots:\n";
    for (const auto& slot : availableSlots) {
        cout << "[" << slot.first << ", " << slot.second << "]\n";
    }

    return 0;
}
```

Work Split:

Algorithm 1:

Pseudocode: Azaan

Code: Omar

Revision: Omar / Azaan

Proving efficiency class: Kevin

Algorithm 2:

Pseudocode: Azaan

Code: Omar

Revision: Omar / Azaan

Proving efficiency class: Kevin

Algorithm 3:

Pseudocode: Azaan

Code: Azaan

Revision: Omar / Azaan

Proving efficiency class: Kevin