Angel Penaloza - angelpenlza@csu.fullerton.edu

# Algorithm 3

**PSEUDOCODE**

```
// a person's scheulde is an array of frees and busies, where the index can be converted to a time
// and vice versa. an empty schedule = [free, free, free, …, free, free, free]
toTime(index)  {
    hours = index * 4
    minutes = (index % 4) * 0.15
    return  hours + minutes
}
toIndex(time) {
    hour = floor(time)
    if(time == hour) return (hour * 4)
    else return (hour * 4) + (time - hour) / 0.15
}
addBusyTime(start_time, end_time) {
    for index in range (toIndex(start_time), toIndex(end_time)):
        schedule[index] = busy
}
addSchedule(schedule) {
// update the daily act for each user, making sure to account for the latest start time
// and earliest end time
    if(group_schedule.dailyAct.startTime < schedule.dailyAct.startTime)
        group_schedule.dailyAct.startTime = schedule.dailyAct.startTime
    if(group_schedule.dailyAct.endTime > schedule.dailyAct.endTime)
        group_schedule.dailyAct.endTime = schedule.dailyAct.endtime
// the original group schedule is empty. after each insertion of a schedule, keeps track of all
// times that are unavailable for everyone in the group
    for time in range (0, len(schedule)):
        if(schedule.time ==  busy)
```

```
                group_schedule.time = busy
}
findMeetingTimes(meeting_length) {
// now that we have all the user's schedules combined to one, we apply the earliest and latest
// times each person is available, adding them as busy times
    group_schedule.addBusyTime(0:00, group_schedule.dailyAct.startTime)
    group_schedule.addBusyTime(group_schedule.dailyAct.endtime, 24:00)
    counter = 0
    meeting_length = toIndex(meeting_length)
// we convert the meeting length, given in time format, to index format. this tells us how many
// free intervals we need to go through to see if a meeting is available
    for index in range(0, len(schedule)):
        if(group_schedule.index == free)
            counter++
        if(group_schedule.index == busy)
            counter = 0;
        if(counter == meeting_length)
            possible_meeting_times.push(toTime(index - meeting_length + 1)
            possible_meeting_times.push(toTime(index + 1)
        if(counter > meeting_length)
            possible_meeting_times.pop()
            possible_meeting_times.push(index + 1)
    for meeting_time in possible meeting_times:
        print(meeting_time)
}
```

**ANALYSIS**

- **toTime(**index**)** and **toIndex(**time**)** both have constant time complexity of O(1). this is because neither of them have loops, they both simply perform mathematical calculations and return the value.

- **addBusyTime(**start_time, end_time**)** has a time complexity of O(i), where i is the length of the interval between the start time and end time. This is because the function will only loop through the given interval and not through anything else. It does not need to go through the entire array to add the busy time interval. This will always be < O(n) because the only way for it to be O(n) is if the user enters a start time of 0:00 and end time of 24:00, which is unrealistic. This is the worst case scenario. Best case scenario is the user adding a 15 minute busy time interval, because the loop would only need to change one value then stop.
- **addSchedule(**schedule**)** has a time complexity of O(n), because it loops through the entire array, comparing each time interval, and adding any necessary busy times. It has two if statements before the for loop which do not affect the time complexity. The best and worst case scenarios are both O(n).
- **findMeetingTimes(**meeting_length**)** also has a time complexity of O(n) because it needs to loop through the entire array to see if there are any overlapping free times. Since all the busy times have already been added, the array only needs to loop through the array one time, all the way through to see where the overlapping intervals are.
- **Overall Use:** each user creates their own schedule, adding busy times as needed. When adding their dailyAct, this is a constant time complexity, since this is just defining a variable. So this, at worst, will be O(n) if their entire day is busy, and more likely than not be < O(n). To add a schedule, we have to add  each person's schedule, which means the time complexity is p * O(n), where p is the number of people in the group. To find the free time intervals, we start by applying the dailyAct, which would have the same complexity as adding a busy time schedule, so < O(n). Then, we loop through the array once, which is O(n). Then to print the values, which isn't mentioned as a function, it is O(s), where s is the size of the array which stores the available times. At its worst case scenario, it's O(n), if all the intervals are very small. It will more than likey be < O(n). So in conclusion, assuming worst case scenarios, we have O(n) + [p * O(n)] + [O(n) + O(n)] + O(n), all added since none of the O(n) functions call any of the other O(n) functions within them. This means that we can evaluate this as O(n) time complexity.