Qasim Hussain, Sumiran Mishra, Alex Jun, Max de la Torre
CPCS 335 Algorithm Engineering
Prof. Kavil Jain
October 1, 2024

*Algorithm 1: Connecting Pairs of Persons (Sumiran)
*Algorithm 2: Greedy Approach to Hamiltonian Problem (Qasim)
*Algorithm 3: Ensuring Convenient Schedules (Alex and Max)

Algorithm 1: Connecting Pairs of Persons

Pseudocode:

- Start at the beginning of the array.
- Check if the adjacent element is one of the pairs
- If so move on to the adjacent to the next element
- If not find an element in the array that is less than or more than one
- Swap elements

Python Implementation:

```
def let_partner_hold_hands(array):
   k = []
   for i in range(len(array)-1):
#check if the adjacent element is one below or after
       if abs(array[i]-array[i+1])!= 1:
           for j in range(i+2, len(array)-1):
               if abs(array[j]-array[i])==1:
                   array[i+1], array[j] = array[j], array[i+1]

           # Add an element in our switch array
           k.append(i+1)
       else:
           i = i + 1
   return k
# test case
test_arr = [0,2,1,3]
print(let_partner_hold_hands(test_arr))
```

Qasim Hussain, Sumiran Mishra, Alex Jun, Max de la Torre
CPCS 335 Algorithm Engineering
Prof. Kavil Jain
October 1, 2024

Efficiency analysis

```
def function (arr):
    K = [ ]                                                          O(1)
    for i in range(len(array)-1):                                    O(n)
        if abs(array[i]-array[i+1]) != 1:                            O(1)
            for j in range (i+2, len(array)-1):                      O(n-2)
                if abs(array[j]-array[i]) == 1:                      O(1)
                    array[i+1], array[j] = array[j], array[i+1]      O(1)

            K.append (i+1)                                           O(1)

    return K
```

=> Tota effeciency

↳ $O(1) + O(n) \cdot O(1) \cdot O(n-2) \cdot O(1) \cdot O(1) + O(1)$

$\Rightarrow O(1) + O(n^2-2n) + O(1)$

$\Rightarrow O(n^2-2n)$

proving efficcency :-

$$\lim_{n\to\infty} \frac{n^2-2n}{n^2} = \lim_{n\to\infty} 1 - \frac{2}{n} \approx 1 - 0 = 1$$

since the limit converges the algorithm has $O(n^2)$ efficiency.

Qasim Hussain, Sumiran Mishra, Alex Jun, Max de la Torre
CPCS 335 Algorithm Engineering
Prof. Kavil Jain
October 1, 2024

Algorithm 2: Greedy Approach to Hamiltonian Problem

Pseudocode

1. Function: find_preferred_starting_city(city_distances, fuel, mpg)
   Input:
      - city_distances: an array of distances between consecutive cities
      - fuel: an array of available fuel at each city
      - mpg: miles per gallon the car can travel with 1 unit of fuel
   Output:
      - index of the preferred starting city

2. Initialize variables:
   - n = length of city_distances (number of cities)
   - total_fuel_needed = 0
   - total_fuel_available = 0
   - current_fuel = 0
   - start_city = 0

3. Iterate through each city (index i from 0 to n-1):
   - fuel_from_city_i = fuel[i] * mpg (total miles the car can drive from fuel at city i)
   - fuel_needed_to_next_city = city_distances[i] (miles needed to reach the next city)
   - current_fuel = current_fuel + fuel_from_city_i - fuel_needed_to_next_city
   - If current_fuel < 0:
      - Set start_city = i + 1 (we cannot start from this city, so move to the next one)
      - Reset current_fuel to 0
   - total_fuel_needed = total_fuel_needed + fuel_needed_to_next_city
   - total_fuel_available = total_fuel_available + fuel_from_city_i

4. If total_fuel_available < total_fuel_needed:
   - Return -1 (no valid starting city exists, but the problem guarantees one)

5. Return start_city

Explanation: We need to find the city where we can start and visit all cities, returning to the starting city with enough fuel.

Greedy approach: Traverse through each city and track the fuel availability and fuel requirements. If at any point the car doesn't have enough fuel to proceed, mark the next city as a potential starting point. Once the loop finishes, the last start_city will be the valid starting city.

Qasim Hussain, Sumiran Mishra, Alex Jun, Max de la Torre
CPCS 335 Algorithm Engineering
Prof. Kavil Jain
October 1, 2024

Time Complexity: The algorithm iterates through all the cities exactly once, making it an O(n) algorithm, where n is the number of cities. Since the document requires the efficiency class, this is O(n).

Python Implementation

```
def find_preferred_starting_city(city_distances, fuel, mpg):
    n = len(city_distances)
    total_fuel_needed = 0
    total_fuel_available = 0
    current_fuel = 0
    start_city = 0

    for i in range(n):
        # Calculate the miles the car can travel from the fuel at city i
        fuel_from_city_i = fuel[i] * mpg
        # Calculate the distance to the next city
        fuel_needed_to_next_city = city_distances[i]

        # Update current fuel after moving to the next city
        current_fuel += fuel_from_city_i - fuel_needed_to_next_city

        # If current fuel is negative, we cannot start from this city
        if current_fuel < 0:
            start_city = i + 1
            current_fuel = 0

        # Track total fuel availability and total fuel needed
        total_fuel_needed += fuel_needed_to_next_city
        total_fuel_available += fuel_from_city_i

    # If the total available fuel is less than the total needed, return -1 (no valid starting point)
    if total_fuel_available < total_fuel_needed:
        return -1

    # Return the preferred starting city
    return start_city

# Sample Input
city_distances = [5, 25, 15, 10, 15]
fuel = [1, 2, 1, 0, 3]
mpg = 10

# Call the function
```

```
preferred_starting_city = find_preferred_starting_city(city_distances, fuel, mpg)
print(f"The preferred starting city is city {preferred_starting_city}")
```

Algorithm 3: Ensuring Convenient Schedules

Pseudocode:

```
Function: find_meeting_slots(busy_schedules, working_periods, duration)
Input:
  busy_schedules: 2D array of busy times for each person
  working_periods: List of earliest and latest available times for each person
  duration: Minimum meeting duration in minutes
Output:
  List of available meeting slots in HH:MM format

Set earliest_start = latest available start time from working_periods
Set latest_end = earliest available end time from working_periods
Combine and sort all busy schedules into combined_busy

Find free slots:
  Initialize last_end = earliest_start
  For each interval in combined_busy:
    If there's enough gap between last_end and interval start, save it as a free slot
    Update last_end to the max of last_end and interval end

Check for a final free slot between last_end and latest_end

Return the free slots in HH:MM format
```

Python Implementation:

```
def convert_time_to_minutes(time_str):
    hours, minutes = map(int, time_str.split(':'))
    return hours * 60 + minutes

def convert_minutes_to_time(minutes):
    hours = minutes // 60
    mins = minutes % 60
```

```
    return f"{hours:02d}:{mins:02d}"

def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []
    for interval in intervals:
        if not merged or interval[0] > merged[-1][1]:
            merged.append(interval)
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])
    return merged

def find_free_intervals(busy_intervals, daily_bounds):
    earliest, latest = daily_bounds
    busy_intervals = [[earliest, earliest]] + busy_intervals + [[latest, latest]]
     busy_intervals = merge_intervals(busy_intervals)
    free_intervals = []
    for i in range(1, len(busy_intervals)):
        start_free = busy_intervals[i - 1][1]
        end_free = busy_intervals[i][0]
        if start_free < end_free:
            free_intervals.append([start_free, end_free])
    return free_intervals

def intersect_intervals(intervals_a, intervals_b):
    i, j = 0, 0
    intersection = []
    while i < len(intervals_a) and j < len(intervals_b):
        start = max(intervals_a[i][0], intervals_b[j][0])
        end = min(intervals_a[i][1], intervals_b[j][1])
        if start < end:
            intersection.append([start, end])
        if intervals_a[i][1] < intervals_b[j][1]:
            i += 1
        else:
            j += 1
    return intersection

def find_common_available_times(schedules, daily_active_periods, meeting_duration):
    free_times_list = []
    for schedule, daily_bounds in zip(schedules, daily_active_periods):
        busy_intervals = [[convert_time_to_minutes(s), convert_time_to_minutes(e)] for s, e in
schedule]
```

```
        daily_bounds = [convert_time_to_minutes(daily_bounds[0]),
convert_time_to_minutes(daily_bounds[1])]
        free_intervals = find_free_intervals(busy_intervals, daily_bounds)
        free_times_list.append(free_intervals)
    common_available_times = free_times_list[0]
    for i in range(1, len(free_times_list)):
        common_available_times = intersect_intervals(common_available_times, free_times_list[i])
    meeting_duration_minutes = meeting_duration
    result = []
    for start, end in common_available_times:
        if end - start >= meeting_duration_minutes:
            result.append([convert_minutes_to_time(start), convert_minutes_to_time(end)])
    return result

# Sample Input
person1_schedule = [['7:00', '8:30'], ['12:00', '13:00'], ['16:00', '18:00']]
person1_daily_act = ['9:00', '19:00']
person2_schedule = [['9:00', '10:30'], ['12:20', '13:30'], ['14:00', '15:00'], ['16:00', '17:00']]
person2_daily_act = ['9:00', '18:30']
meeting_duration = 30

# Function Call
schedules = [person1_schedule, person2_schedule]
daily_active_periods = [person1_daily_act, person2_daily_act]
available_times = find_common_available_times(schedules, daily_active_periods,
meeting_duration)

print(available_times)
```

Time Complexity: For each of the N persons, their M busy intervals are sorted. Sorting each list of intervals takes O(MlogM) time. Therefore, the time complexity is O(N*MlogM).