CSU33014 Final Code Explanation

1. **less_redundant_number_within_k_degrees(struct person *start, int total_people, int k)**

   This function is similar to the original *number_within_k_degrees()* function except a few important changes. I changed the variable reachable from an array of Boolean values to an array of integer values called *steps_away*. This array will now store the number of steps each person is away from the start person. So for example if a person with index 205 is 4 steps away from the start person, *steps_away[205]* will now contain 4. After all the values in reachable are set to 0, the new recursive function is called with the *steps_remaining* parameter as *k* and also the steps parameter as *k*. As the recursive function iterates the *steps_remaining* variable will be decremented whereas the steps variable will remain the same (this will be explained further in the recursive function explanation). When the recursive function is finished its iterations, the *steps_away* array will now contain the distances of the people that could be reached within the required steps from the start person. The function now searches through the *steps_away* array to see what values are not equal to 0 and then increments a count. The count represents the number of people within *k* degrees from the start person.

2. **less_redundant_find_reachable_recursive(struct person *current, int steps_remaining, int *steps_away, int steps):**

   I based this function off the original given *find_reachable_recursive()* function but with some optimisations to remove the redundancy and speed up the operation. The function now takes an extra parameter called *steps*. This *steps* variable remains the same from start to finish as it stores the *k* variable (i.e. the number of steps away from the start person in which we can search). Using the steps value, the *steps_remaining* value is taken away from *steps* and the result is stored in the *steps_away* array at the index of the current person. As the recursive function iterates, the *steps_remaining* value is decremented so the function stops once it reaches 0. Since each person contains a variable *number_of_known_people*, this could be used as the number of iterations for searching through the current person's neighbours. The function now searches the neighbours of the current person to check how many steps away they are from the start person and stores the result as *steps_away_temp*. The function calls the recursive function again with a decremented *steps_remaining* only if *steps_away_temp* is either equal to 0 to greater than the *steps – steps_remaining* (i.e. if the steps away of the current person is less than the steps away of their neighbour). This stops the function from redundantly visiting the same person of the graph again and again.

3. **Parallelisation**
   To parallelise the **less_redundant_find_reachable_recursive()** function, I made use of OpenMp. No changes were made to how the function works except for the inclusion of some pragma statements. I made use of the opt_set_num_threads() function to define the number of threads used by openmp. I decided to set num_known threads the for loop searches num_known times. I then placed a pragma statement before the for loop in order to parallelise it.