FEYZI CAN ESER – BRIEF EXPLANATION OF PROGRAM

- **Program Structure:**
  - Main.c reads the particle information, allocates memory for the particle and collision array, implements the main loop, and prints out all particle information before freeing up all dynamically allocated memory
  - Collisions.h includes declarations for the functions defined in Collisions.c, which handle all aspects of updating particles and collisions. Collisions.h also declares the global variables initialized in Main.c
- **Key Elements Used:**
  - **Structs:** Collision and Particle structs hold the information specified in the prompt
  - **Arrays:** Arrays of pointers to collision and particle structs with dynamically allocated memory
  - **File Input**: Reads initial particle data from a specified file.
  - **Dynamic Memory Allocation**: Arrays for particles and potential collisions are dynamically allocated, as well as the particle and collision structs themselves
  - **Collision Detection**: At t=0, calculate all potential collision times initially, sort them, and process the earliest collision each time
  - **Event-Driven Simulation**: Event-driven approach to handle collisions and update particle states efficiently
    - Only particles affected by a collision see their collision times updated.
  - **Final Position Update**: At the simulation end time, I updated all particle positions and outputted the final data.

WHY IT WORKS I: MAIN PROGRAM FLOW

Program Flow in Main.c

1. Read particle information from file and create particles_array
   1. Particles_array filled using the createParticle() func, which dynamically allocates memory for a single particle and returns a pointer to the struct filled with correct initial positions, velocities, and collision counts set to 0
2. Collisions_array is filled by calling calculate_all_particle_collisions(), which appends all possible particle-particle collisions, and then by calling calculate_all_wall_collisions(), which appends all possible particle-wall collisions for each of the 4 walls
   1. Thus, the collisions_array includes all possible collision combinations

```c
// Loop until end time is reached
while (t_current < t_end) {
    // Sort collisions by time
    sort_collisions(collisions_array);
    // Get next collision
    Collision * next_collision = collisions_array[0];

    // Update time and break if needed
    t_current = next_collision->time;
    if( t_current > t_end){ break;}

    // Process collisions - update the particle's position and their velocities
    process_collision(next_collision);
    // Update affected particles' collisions
    Particle * particle_1 = next_collision->particle_1;
    Particle * particle_2 = next_collision->particle_2;
    update_affected_particles(particle_1, particle_2, collisions_array);
}
```

3. Main Loop:

1. sort_collisions(): sort array in place using insertion sort
2. Select first element in the sorted collision array, update the current time to that collision's time – break if end time exceeded
3. Process_collision(): Update kinetic energies of the particle(s) accordingly
4. Update_affected_particles(): Loop through the collisions_array, update the times of collisions including either of the two particles (particle 2 is null if we just had a particle-wall collision)
   1. Function terminates early when the expected number of collisions is updated

```c
// Update particles to final positions
for (int i=0; i<n_particles; i++){
    Particle * particle = particles_array[i];
    particle->xpos = particle->xpos + particle->xvel * (t_end - particle->lastUpdateTime);
    particle->ypos = particle->ypos + particle->yvel * (t_end - particle->lastUpdateTime);
}
```

4. Update all particle's positions to the end time positions before printing final outputs and freeing the memory

WHY IT WORKS II: DESIGN NOTES

Correctness of Results:

- Compared the outputs of program to the model outputs of the test files
- Used printf() to print intermediate results to ensure each step was correct (print statements since removed)

Modular Design:

- Built the program in a modular fashion, making it easier to test, debug, and extend upon

Unit Testing:

- Performed tests on individual functions to ensure they operated correctly in isolation

Error handling:

- Put checks to raise flags when any dynamic memory allocation failed

Making functions robust to errors:

- **Collision times:** Collision_time is set to end_time +1 for infeasible collisions (those that appear to be in the past)
- **Particle structs:**
  - Correctly updating the last update time in Particle structs ensures that any extraneous update operations in the same timestep do not change anything
  - Particle structs have a variable wall: 0 for no walls involved, 1 for left, 2 for right, 3 for top, and 4 for bottom walls
    - Collision structs have particle_2 pointer set to Null for wall collisions - ensuring that each collision is handled appropriately
- **Current time:** A global variable is used to keep track of current time, making sure every function uses up-to-date information