```python
import enum
import itertools
from enum import Enum
from typing import List


Direction = Enum("Direction", "RIGHT LEFT")


class Particle:
    """A particle object"""

    def __init__(self, position: int, direction: Direction):
        self.position = position
        self.direction = direction

    def change_direction(self):
        """Change direction of a particle"""
        self.direction = Direction.LEFT if self.direction == Direction.RIGHT else Direction.RIGHT

    def change_position(self, steps: float):
        """Move a particle position"""
        self.position = self.position + \
            steps if self.direction == Direction.RIGHT else self.position - steps


class Pole:
    """
    A 1D line of length N cm.
    """

    def __init__(self, length: int, speed: int, particles: List[Particle]):
        self.length = length
        self.speed = speed
        self.particles = particles
        self.removed_particles = []
        # times_of_particles_removal should be an array of equal length to particles
        self.times_of_particles_removal = [-1 for _ in particles]

    def is_particle_to_be_removed(self, idx: int):
        return self.particles[idx].position > self.length or self.particles[idx].position < 0

    def is_particle_removed(self, idx: int):
        """
        Check if particle is removed from the pole

        Parameters
        ----------
        idx : int
            Index of particle
        """
        return self.particles[idx] in self.removed_particles

    def are_particles_in_same_position(self, indices: List[int]):
        """
        Check if particles are in same position

        Parameters
        ----------
        indices : List[int]
            List of indices of particles
        """
        return all([self.particles[i].position == self.particles[indices[0]].position for i in indices])

    def are_particles_moving_in_same_direction(self, indices: List[int]):
        """
        Check if particles are moving in same direction

        Paramerters
        ----------
        indices : List[int]
            List of indices of particles
        """
        return all([self.particles[i].direction == self.particles[indices[0]].direction for i in indices])

    def move_particles(self, idx: int, steps: float, current_time: int):
        """
        Move a particle on the pole

        Parameters
        ----------
        idx : int
            Index of particle
        steps : float
            Number of steps to move a particle
        current_time : int
            Current time of simulation of motion
        """
        self.particles[idx].change_position(steps)
        if self.is_particle_to_be_removed(idx):
            self.removed_particles.append(self.particles[idx])
            # overide to place particle at this position
            self.times_of_particles_removal[idx] = current_time

    def simulate(self, current_time: int):
        """
        Start motion of particles on the pole

        Parameters
        ----------

        current_time : int
            Current time of simulation of motion
        """
        mini_steps = 0.5
        for _ in range(int(self.speed / mini_steps)):
            i = 0
```

```
            while i < len(self.particles):
                if not self.is_particle_removed(i):
                    num_of_particles_changed = 0
                    try:
                        while self.are_particles_in_same_position([i, i + num_of_particles_changed + 1]):
                            if not self.are_particles_moving_in_same_direction([i, i + num_of_particles_changed + 1]):
                                num_of_particles_changed += 1
                                self.particles[i +
                                               num_of_particles_changed].change_direction()
                                self.move_particles(
                                    i + num_of_particles_changed, mini_steps, current_time)
                    except IndexError:
                        pass
                    if num_of_particles_changed > 0:
                        self.particles[i].change_direction()
                    self.move_particles(i, mini_steps, current_time)
                    i += num_of_particles_changed
                i += 1


class World:
    """
    A world(simulation) that contains a number of poles with particles on them.
    """

    def __init__(self, poles: List[Pole]):
        self.poles = poles
        self.removed_poles = {}
        self.current_time = 0

    def simulate(self):
        """
        Simulate the world where the particles are removed from the Poles as they move to either end.
        """
        while len(self.poles) > len(self.removed_poles.keys()):
            yield self.current_time
            self.current_time += 1
            for i, pole in enumerate(self.poles):
                if i not in self.removed_poles.keys():
                    pole.simulate(self.current_time)
                    if len(pole.particles) == len(pole.removed_particles):
                        idx = self.poles.index(pole)
                        self.removed_poles[idx] = pole
        yield self.current_time


def get_direction_permutations(number_of_particles: int):
    """
    Get all possible permutations of directions for a given number of particles

    Parameters
    ---------
    number_of_particles: int
        Number of particles
    """
    directions = [Direction.RIGHT, Direction.LEFT]
    permutations = [i for i in itertools.product(
        directions, repeat=number_of_particles)]
    return permutations


def main(length_of_pole: int, speed: int, starting_positions: List[int]):
    """
    Create objects for each pole and simulate the movement of the particles.

    Parameters:
    ----------

    length_of_pole: int
        The length of the pole in cm
    speed: int
        The speed of the particles in cm/s
    starting_positions: List[int]
        The starting positions of the particles in cm
    """
    starting_positions.sort()
    direction_permutations = get_direction_permutations(
        len(starting_positions))
    poles = []

    for permutation in direction_permutations:
        particles = []
        for idx, starting_position in enumerate(starting_positions):
            particles.append(Particle(starting_position, permutation[idx]))
        poles.append(Pole(length_of_pole, speed, particles))

    world = World(poles)
    _ = [t for t in world.simulate()]

    direction_permutations_char = [
        ["R" if j == Direction.RIGHT else "L" for j in i] for i in direction_permutations]
    times_of_particles_removal = []
    for pole in poles:
        times_of_particles_removal.append(pole.times_of_particles_removal)

    first_to_drop_off_time = -1
    last_to_drop_off_time = -1
    for d, t in zip(direction_permutations_char, times_of_particles_removal):
        first_to_drop_off_time = min(t) if first_to_drop_off_time == -1 or min(t) < first_to_drop_off_time \
            else first_to_drop_off_time
        last_to_drop_off_time = max(t) if last_to_drop_off_time == -1 or max(t) > last_to_drop_off_time \
            else last_to_drop_off_time

    first_to_drop = {first_to_drop_off_time: []}
    last_to_drop = {last_to_drop_off_time: []}
    for d, t in zip(direction_permutations_char, times_of_particles_removal):
        if min(t) == first_to_drop_off_time:
```

```
            first_to_drop[first_to_drop_off_time].append(d)
        if max(t) == last_to_drop_off_time:
            last_to_drop[last_to_drop_off_time].append(d)
    print(
        f"\nPermutations with first particle to drop off:  at time {first_to_drop_off_time}")
    print(
        f"Permutations with last particle to drop off:  at time {last_to_drop_off_time}")


def accept_input():
    """Accept user input and begin the simulation"""

    length_of_pole = int(input("Length of pole (cm): "))
    speed = 1
    starting_positions = []
    for i in range(int(input("Number of starting positions: "))):
        starting_positions.append(
            int(input(f"Starting position {i + 1}: (cm) ")))
    main(length_of_pole, speed, starting_positions)


if __name__ == "__main__":
    accept_input()
```