Answer1: Flight Path without Intersection

I will use the `matplotlib` library to visualize the flight paths. This problem primarily focuses on plotting and visualizing paths rather than checking for intersections so that is why I am using python.

Code:

```
import matplotlib.pyplot as plt


def draw_flights(flight_paths):
    """
    Draws flight paths on a 2D plot using matplotlib.


    Args:
    flight_paths (list of lists): Each list contains tuples representing coordinates for a flight path.
    """
    # Create a new figure and axis for plotting
    fig, ax = plt.subplots()


    # Plot each flight path
    for path in flight_paths:
        # Unpack the list of coordinates into x and y lists
        x, y = zip(*path)
        # Plot the path with markers for each point
        ax.plot(x, y, marker='o', linestyle='-', label=f'Flight {flight_paths.index(path) + 1}')
```

```
    # Set the labels and title for the plot
    ax.set_xlabel('X Coordinate')
    ax.set_ylabel('Y Coordinate')
    ax.set_title('Flight Paths')
    # Add a legend to the plot
    plt.legend()
    # Enable the grid for better readability
    plt.grid(True)
    # Show the plot
    plt.show()


def main():
    # Define the flight paths
    flight_paths = [
        [(1, 1), (2, 2), (3, 3)],
        [(1, 1), (2, 4), (3, 2)],
        [(1, 1), (4, 2), (3, 4)]
    ]


    # Draw the flight paths
    draw_flights(flight_paths)


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

Answer 2: Distributing Apples Proportionally

I will use a greedy approach to allocate apples such that the total weight each person receives is close to their proportional share.

Code:

```
def distribute_apples(apples, payments):
    """
    Distributes apples among people proportionally based on the amount they paid.

    Args:
    apples (list of int): Weights of the apples.
    payments (list of int): Amounts paid by each person.

    Returns:
    list of lists: Each sublist contains apples assigned to a person.
    """
    # Calculate total weight of apples
    total_weight = sum(apples)
    # Calculate each person's proportion of the total weight
    proportions = [p / sum(payments) for p in payments]
    # Initialize allocations for each person
    allocations = [[] for _ in range(len(payments))]
    weights = [0] * len(payments)
```

```
    def allocate():
        """
        Allocate apples to each person based on the greedy approach.
        """
        nonlocal apples
        # Sort apples in descending order to allocate larger apples first
        for weight in sorted(apples, reverse=True):
            # Find the person with the least weight so far
            min_index = weights.index(min(weights))
            # Allocate the apple to this person
            allocations[min_index].append(weight)
            weights[min_index] += weight
            apples.remove(weight)

    # Perform the allocation
    allocate()

    # Print the results
    for i, allocation in enumerate(allocations):
        print(f"Person {i+1}: {', '.join(map(str, allocation))}")

def main():
    # Define the apple weights and payments
    apples = [400, 100, 400, 300, 200, 300, 100, 200]
```

```
    payments = [50, 30, 20]  # Ram, Sham, Rahim

    # Distribute the apples
    distribute_apples(apples, payments)

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

Answer 3: Kill All And Return Home

I used a recursive approach with backtracking to explore all possible paths.

Code:

```
def find_paths(soldiers, start_position):
    """
    Finds all unique paths for a castle to kill all soldiers and return to the start.

    Args:
    soldiers (set of tuples): Coordinates of soldiers on the chessboard.
    start_position (tuple): Starting position of the castle.

    Returns:
    list of lists: Each sublist represents a path the castle can take.
    """
    # Directions for movement: Right, Down, Left, Up
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
```

```
all_paths = []

def is_valid(x, y):
    """
    Check if a coordinate is valid (i.e., contains a soldier).

    Args:
    x (int): X coordinate.
    y (int): Y coordinate.

    Returns:
    bool: True if valid, False otherwise.
    """
    return (x, y) in soldiers

def dfs(x, y, path):
    """
    Depth-First Search to find all paths for killing soldiers and returning home.

    Args:
    x (int): Current X coordinate.
    y (int): Current Y coordinate.
    path (list of str): Current path taken by the castle.
    """
```

```
    # Base case: If no soldiers are left, return to the start
    if not soldiers:
        path.append(f"Arrive ({start_position[0]},{start_position[1]})")
        all_paths.append(path[:])
        return

    # Explore each direction
    for dx, dy in directions:
        nx, ny = x, y
        # Move in the current direction until hitting a boundary or a non-soldier cell
        while is_valid(nx + dx, ny + dy):
            nx += dx
            ny += dy

        # If a soldier is found, proceed with this path
        if (nx, ny) in soldiers:
            soldiers.remove((nx, ny))  # Remove the soldier from the set
            path.append(f"Kill ({nx},{ny}). Turn Left")
            dfs(nx, ny, path)
            path.pop()  # Backtrack
            soldiers.add((nx, ny))  # Re-add the soldier for further explorations

# Convert the list of soldiers to a set for efficient operations
soldiers = set(soldiers)
```

```
    # Start DFS from the initial position
    dfs(start_position[0], start_position[1], [])

    return all_paths

def main():
    # Sample Input
    soldiers = [
        (1, 1), (8, 9), (1, 9), (4, 1), (4, 2), (4, 8),
        (2, 6), (5, 6), (8, 2), (5, 9), (2, 8)
    ]
    start_position = (1, 2)

    # Find all valid paths
    paths = find_paths(soldiers, start_position)

    if not paths:
        print("No valid paths found.")
    else:
        print(f"Thanks. There are {len(paths)} unique paths for your 'special_castle'")
        for idx, path in enumerate(paths, 1):
            print(f"Path {idx}")
            print("=======")
            for step in path:
```

```
        print(step)


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