**Problem 2: My Money My Shares**

**Approach to solve that problem :**

1. **Input Collection :**

* Collect apple weights from the user until **’-1’** is entered, which stops the input.
* Store these weights in a vector.

1. **Calculate Target Weights :**

* Calculate the total weight of all apples.
* Determine the target weights for Rahim, Sham, and Ram based on their respective proportions (20%, 30%, and 50%).

1. **Recursive Backtracking :**

* Define three recursive functions to allocate weights to Rahim, Sham, and Ram. These functions attempt to find a combination of apple weights that matches the target weights for each individual.

1. **Rahim's Allocation :**

* Start by trying to allocate apples to Rahim such that their total weight equals 20% of the total weight.
* Use a recursive function (**‘shareForRahim’**) that attempts to pick or not pick each apple, updating the total weight accordingly.
* If a valid combination is found, move on to Sham's allocation.

1. **Sham's Allocation :**

* Use a similar approach (**‘shareForSham’**) to allocate the remaining apples to Sham such that their total weight equals 30% of the total weight.
* If a valid combination is found, move on to Ram's allocation.

1. **Ram's Allocation :**

* Finally, allocate the remaining apples to Ram using (**‘shareForRam’**) so that their total weight equals 50% of the total weight.
* If a valid combination is found, print the distribution.

1. **Base Conditions :**

* Each recursive function has a base condition to check if the target weight is reached.
* If the target weight is reached for Rahim, proceed to Sham's allocation.
* If the target weight is reached for Sham, proceed to Ram's allocation.
* If the target weight is reached for Ram, print the distribution and exit.
* If the distribution is not possible, print **“No valid distribution found”**.

**Help from internet to solve this problem :** To tackle this problem, I watched relevant videos from Pepcoding's "Recursion & Backtracking - Level 1" and "Backtracking - Level 2" playlists. The most beneficial videos for solving this problem were:

1. **Video No. 21 from the "Backtracking - Level 2" playlist :** Combinations in Cpp

Recursive | Recursion Interview Problems | Backtracking

* **YouTube Link :**Combinations in Cpp Recursive | Recursion Interview Problems | Backtracking

1. **Video No. 22 from the "Backtracking - Level 2" playlist :** How to Permute in Cpp

- Part 2 | Permutations and Combinations Algorithm

* **YouTube Link :** How to Permute in Cpp - Part 2 | Permutations and Combinations Algorithm

These resources provided invaluable insights for addressing the problem.

**Problem 3: Kill All And Return Home**

**Approach to solve that problem :**

1. **Initialize the Chessboard:**

* Read the number of soldiers and their coordinates
* The chessboard is represented as a 10x10 grid. The grid is initialized with zeros. Soldiers are placed on the board by setting the corresponding grid cells to 1.
* The castle moves on this grid and can change direction based on certain conditions..

1. **Set the Starting Point:**

* Read the starting coordinates of the castle.
* Initialize a visited array to track visited positions and avoid cycles.

1. **Defining the Move Rules:**

* **Kill :** If the castle encounters a soldier (i.e., a cell with value 1), it can kill the soldier, which changes the cell value to 0 temporarily, and continues in a new direction (left turn).
* **Jump :** If the castle encounters a soldier (i.e., a cell with value 1), it can jump over a soldier and continue in the same direction.
* If the castle does not encounter a soldier (i.e., a cell with value 0), continue moving in the current direction.

1. **Base Cases:**

* **Negative Base Case :** If the castle moves out of bounds, the recursion stops.
* **Positive Base Case :** If the castle reaches the destination coordinates, the path is added to the result list.

1. **Recursive Movement:**

* The function solve is a recursive method that explores all possible moves from the current position **(‘sr’, ‘sc’)** to the destination **(‘dr’, ‘dc’)**. The method updates the path so far **(‘psf’)** and tracks visited cells to avoid infinite loops.

1. **Handling Directions:**

* The castle can move in four directions **:**
* 'D' (Down)
* 'R' (Right)
* 'T' (Top)
* 'L' (Left)

Based on the current direction, the castle either kills or jumps over a soldier and adjusts its path accordingly Cpp

**Help from internet to solve this problem :** To solve this problem, I watched relevant videos from Pepcoding's "Recursion & Backtracking - Level 1" and "Backtracking - Level 2" playlists, as well as consulted the "Rat in a Maze" problem on GeeksForGeeks. The most useful resources were:

1. **Video No. 31 from the "Recursion & Backtracking - Level 1" playlist :** Get Maze Paths - Solution | Recursion | Data Structures and Algorithms in Cpp

* **YouTube Link :** Get Maze Paths - Solution | Recursion | Data Structures and Algorithms in Cpp

1. **Video No. 33 from the "Recursion & Backtracking - Level 1" playlist :** Get Maze Paths with Jump - Solution | Recursion | Data Structures and Algorithms in Cpp

* **YouTube Link :** Get Maze Paths with Jump - Solution | Recursion | Data Structures and Algorithms in Cpp

1. **GeeksForGeeks :** Rat in a Maze

* **Link :** <https://www.geeksforgeeks.org/rat-in-a-maze/>

**Problem 1: No accident, please**

**Approach to solve that problem :**

* Represent each flight path as a list of coordinates.
* **Example :** Flight 1: **‘[ (1, 1), (2, 2), (3, 3) ]’**, Flight 2: **‘[ (1, 1), (2, 4), (3, 2) ]’**, Flight 3: **‘[ (1, 1), (4, 2), (3, 4) ]’**.
* For each pair of flight paths, check if they intersect at any point other than the starting point.
* If they do, modify the paths to avoid intersection. This could involve rerouting one of the paths to a nearby coordinate that is not occupied.
* Use the **‘Bentley–Ottmann’** algorithm to detect if two line segments intersect.
* Optimise the rerouting process using **‘Dijkstra's algorithm’** to ensure minimal deviation from the original path while avoiding intersections.
* Use a graph plotting library to visualise the flight paths. Each flight path will be plotted in a different colour to distinguish them.

**Help from internet to solve this problem :** Learned about the **‘Bentley–Ottmann’** algorithm from **ChatGPT** but didn't complete my study of it due to time constraints.