**Operation 1 Join a game:**

Join a game

Input game code

IF (Game code exists) => Go to server

ELSE => Invalid game code => Go to home page

**Operation 2 Calculate who owes who:**

**1. Extract Input Data**

* **Step 1:** Extract the input string (either entered by the user or game data from the app).
* **Step 2:** **Condition Check:** Is the input string valid?
  + **IF** valid → Proceed to the next step.
  + **ELSE:** Correct the invalid data and return to the validation step.

**2. Organize Data**

* **Step 1:** Create a dictionary with:
  + **Key:** playerName
  + **Value:** PlayerStats (containing BuyIn and BuyOut data for each player).
* **Step 2:** **Condition Check:** Does the total BuyIn match the total BuyOut?
  + **IF** they match → Proceed to the next step.
  + **ELSE:** Suggest possible solutions to fix discrepancies and return to the validation step.

**4. Prepare for Settling Debts**

* **Step 1:** Create two lists from the dictionary:
  + **Losers:** Players with a net loss (negative balance).
  + **Winners:** Players with a net profit (positive balance).
* **Step 2:** Sort both lists by the absolute values of profit/loss (descending order).

Sort Who Owes who in lists

**Equal Pays**

**Step 1:** The **User/Caller** triggers the function to process debts.

**Step 2:** The system loops through the **losersList** and, for each loser:

* The system starts another loop to check against the **winnersList**.

**Step 3:** For each pair (loser and winner):

* If the loser owes an amount equal to what the winner is owed, this triggers:
  + Printing a payment statement (via PrintWhoOwesWhoSentence).
  + Marking the corresponding loser and winner as "paid" by setting their values to 0.
  + Removing the loser and winner from their respective lists.

**Step 4:** Once all losers are processed:

* If any unpaid losers remain, the system checks if further payments can be calculated and prints an empty line to separate payments (if necessary).

**Step 5:** The system finishes processing and returns control to the caller.

**DFS**

**1. Iterate Over Winners**

* **Step 1:** Loop through the list of winners.
* **Step 2:** **Condition Check:** Are there winners to process?
  + **IF** yes → Continue with the loop.
  + **ELSE** → Exit the process (no more winners).

**2. Reorder Losers and Winners**

* **Step 1:** Reorder the losersList:
  + Start with the **smallest** values (ascending order).
* **Step 2:** Reorder the winnersList:
  + Start with the **largest** values (descending order).

**3. Search for "Perfect Matches"**

* **Step 1:** Use **DFS** to find a combination of losers that matches the current winner’s value exactly:
  + Use a helper method (e.g., DFSettleWinner) to find potential combinations.
* **Step 2:** **Condition Check:** Is a "perfect match" found?
  + **IF** yes → Store the combination in foundCombo and proceed.
  + **ELSE** → Skip to the next winner.

**4. Process "Perfect Matches"**

* **Step 1:** For each loser in the foundCombo:
  + Print the settlement details (who owes whom and how much).
  + Remove the loser from the losersList.
* **Step 2:** Remove the winner from the winnersList.

**5. Adjust Index**

* **Step 1:** **Condition Check:** Was a "perfect match" found?
  + **IF** yes → Adjust the index (i--) to recheck the next winner.
  + **ELSE** → Proceed to the next winner.

**6. Repeat or Finish**

* **Step 1:** Go to the next winner in the winnersList and repeat the process.
* **Step 2:** Finish when all winners are processed.

**Most to Last**

**1. Prioritize Favorite Players**

* **Step 1:** Use PrioritizeFavPlayers to sort the losersList and winnersList so that favored players (as defined in favPlayers) are prioritized for payments or collections.

**2. Sort Remaining Players**

* **Step 1:** Sort remaining players in the following manner:
  + **Biggest Loser** pays the **Biggest Winner** first.
* This ensures that large debts are settled early in the process.

**3. Iterating Over Losers and Winners**

* **Step 1:** Iterate through all losers and, within that loop, iterate through all winners.
* **Condition Check:** If the winner's value is 0 (already paid), skip to the next winner.

**4. Transfer Money**

* Use the following logic:
  + Compare the loserPayAmount (how much the loser can pay) and the winnerPayAmount (how much the winner needs to receive).

**5. Handling Specific Cases**

**Case 1: Loser Pays Less Than Winner Needs**

* **Condition Check:**
  + If the loserPayAmount is less than winnerPayAmount, deduct loserPayAmount from the winner's debt.
* **Special Condition for Small Balances:**
  + If the remaining balance for either party is less than the minTransfer threshold, adjust the amount to ensure meaningful transfers in the next iterations.

**Case 2: Loser Pays More Than Winner Needs**

* **Condition Check:**
  + If the loserPayAmount is more than winnerPayAmount, deduct the winner’s required amount from the loser’s available funds.
* **Special Condition for Small Balances:**
  + If the remaining balance would violate the minTransfer threshold, adjust the payment accordingly to leave a balance that can be settled meaningfully later.

**6. Updating Data Structures**

* After each payment:
  + Update the winnersList and losersList with the new balances.
  + If a "partial" payment is made, move to the next winner/loser as needed.

**7. Return Final Output**

* After all payments are processed, return the printCashIndex, which tracks the details of each transaction.

**Operation 3 Pause game:**

**Preexisting condition**: Player in game.

Clicks Pause.

IF (player admin) => Game pauses

ELSE => Player not admin => Toast(“Can’t pause game, not admin.”)