Mind Games

1 Classes and Methods

1.1 CalculateProbability Class

The `CalculateProbability` class is used to compute the probability of winning or losing for Alice and Bob using dynamic programming. It maintains the state of the `dp` variable. It also holds the global variable of the max wins that Alice or Bob should achieve.

1.1.1 Initial Condition

$$\texttt{self.dp} = \begin{cases} 1, & \text{if } (\texttt{alice\_wins}, \texttt{bob\_wins}) = (\texttt{max\_alice\_wins}, \texttt{max\_bob\_wins}) \\ -1, & \text{otherwise} \end{cases}$$

self.dp holds the probability of Alice achieving **alice_wins** wins and Bob achieving **bob_wins** wins.

1.1.2 calc_dp Method

The `calc_dp` calculates the probability recursively using dynamic programming. If the number of wins for Alice or Bob exceeds the maximum, it returns 0. Otherwise, it checks whether the value is already computed. If not, it calculates the probability using the following recurrence relation:

`dp[alice_win][bob_wins]` =

$\left(\frac{\text{alice\_wins}}{\text{alice\_wins+bob\_wins}}\right) \times \text{calc\_dp(alice\_wins, bob\_wins+1)} + \left(\frac{\text{bob\_wins}}{\text{alice\_wins+bob\_wins}}\right) \times \text{calc\_dp(alice\_wins+1, bob\_wins)}$ The computed value is then stored in the `dp` table for future use. This recursion relation is based on the fact that only 2 cases are possible at every junction and the value calculated from any route will be unique, hence if it has already been calculated, it can be returned without any further processing.

1.2 PlayerBase Class

The `PlayerBase` class models the basic behavior of the players, Alice and Bob. It gives them the methods to handle outcomes of games and initializes their

1.2.1 Constructor

The constructor `__init__` initializes the player with the following attributes:

- `past_play_styles`: A list storing the past moves.
- `results`: A list storing the results of past games, initialized based on whether the player is Alice or Bob.
- `points`: The current score of the player, initialized to 1.

- `opp_points`: The current score of the opponent, initialized to 1.

1.2.2 observe_result Method

The `observe_result` method updates the player's points and the opponent's points based on the result of the current round. It also appends the player's and the opponent's moves to their respective histories.

2 Game Simulation

Several functions are used to simulate the rounds of the game between Alice and Bob.

2.1 update_payoff_matrix Function

The $\texttt{update}_{p}ayoff\_matrixfunctionupdatesandreturnsa3x3matrixbasedonAlice'sandBob'spoints.Eachel$
`y, z)`$, where$:

x `is the probability that Alice wins,`

y `is the probability of a draw,`

z `is the probability that Bob wins.`

2.2 determine_win_loss Function

This function determines the result of a round between Alice and Bob. It uses the `payoff_matrix` to select the probabilities based on the moves made by Alice and Bob. A random number is generated to decide whether Alice wins, it is a draw, or Bob wins, based on the probability tuple from the payoff matrix.

2.3 simulate_round Function

The `simulate_round` function simulates a single round of the game. Both Alice and Bob make their respective moves using the `play_move` method (not defined in the provided code). The result of the round is determined using the `determine_win_loss` function, and both players update their points and move history accordingly.

2.4 monte_carlo Function

The `monte_carlo` function simulates multiple rounds of the game using the Monte Carlo method. It runs the `simulate_round` function for the specified number of rounds and prints the points of Alice and Bob after each round. Monte Carlo simulations runs for only 10^5 iterations.

3 Solution for Question 1:

The solution to the 1st question of the Assignment, using dynamic programming to compute the probability of winning for Alice and Bob. This solution is built upon the previously explained classes and functions.
The code provided consists of three main functions:

- `calc_prob()`: Computes the probability of winning for Alice and Bob based on the given `entry_no`.
- `calc_expectation()`: Computes the expected value of the difference in wins between Alice and Bob.
- `calc_variance()`: Computes the variance of the difference in wins between Alice and Bob.

3.1 Formula's Used:

3.1.1 Expected Value Formula

The expectation value $E[X]$ can be written as:

$$E[\sum_{i=1}^{T} X] = \sum_{i=1}^{B} P(x_i) \cdot (B - 2 \cdot b_i)$$

Where:
- $P(x_i)$ is the probability for the i-th Round.
- B is the upper limit of Bob's wins.
- b_i is the number of Bob's wins in the i-th Round.

3.1.2 Variance Formula

The variance $\mathrm{Var}(X)$ is given by:

$$\mathrm{Var}(\sum_{i=1}^{T} X) = \sum_{i=1}^{B} P(x_i) \cdot (B - 2 \cdot b_i)^2 - (E[X])^2$$

Where: - $P(x_i)$ is the probability for the i-th scenario.
- B is the upper limit of Bob's wins.
- b_i is the number of Bob's wins in the i-th scenario.
- $E[X]$ is the expectation value computed above.

Derivation of the Expected Value Formula

Let the random variable X_i be defined as follows:

$$X_i = \begin{cases} 1, & \text{if Alice wins round } i \\ 0, & \text{if the round is a draw} \\ -1, & \text{if Alice loses round } i \end{cases}$$

Step 1: Definition of Expected Value

The expected value $E[X]$ of a random variable X is defined as the sum of each possible outcome multiplied by its probability:

$$E[X] = \sum_{i=1}^{n} P(x_i) \cdot X_i$$

Where $P(x_i)$ is the probability that outcome i occurs, and X_i represents the result of Alice's performance in each round (either a win, draw, or loss).

Step 2: Adjusting for Bob's Wins

Given that Alice's win/loss outcome is related to Bob's performance, we adjust the expected value calculation to incorporate Bob's wins in each round. The expression $(B-2\cdot b_i)$ is used to account for Bob's total wins and losses: - B is the upper limit of Bob's wins (total number of rounds Bob could win). - b_i represents the number of Bob's wins in the i-th round. The term $(B - 2 \cdot b_i)$ adjusts the number of Bob's wins, implying some relation to Alice's results or a game where each win of Bob counts negatively for Alice's success.

Thus, the expected value formula becomes:

$$E[\sum_{i=1}^{T} X] = \sum_{i=1}^{B} P(x_i) \cdot (B - 2 \cdot b_i)$$

Step 3: Final Expected Value Formula

The formula $E[X] = \sum_{i=1}^{B} P(x_i) \cdot (B - 2 \cdot b_i)$ gives us the expected value, adjusted by Bob's performance.

Derivation of the Variance Formula

Step 1: Definition of Variance

The variance $\text{Var}(X)$ of a random variable X is defined as the expected value of the squared deviations from the mean:

$$\text{Var}(X) = E[X^2] - (E[X])^2$$

Where: - $E[X^2]$ is the expected value of X^2, which represents the squared outcomes of the random variable X. - $(E[X])^2$ is the square of the expected value calculated earlier.

Step 2: Squaring the Deviation Terms

We already have $X_i = B - 2 \cdot b_i$. To calculate the variance, we need to square this expression for each outcome:

$$X_i^2 = (B - 2 \cdot b_i)^2$$

This represents the squared deviation from the expected outcome for each round.

Step 3: Calculating Expected Value of Squared Deviations

Next, we compute the expected value of the squared deviations by summing over all possible outcomes, weighted by their probabilities:

$$E[\sum_{i=1}^{T} X^2] = \sum_{i=1}^{B} P(x_i) \cdot (B - 2 \cdot b_i)^2$$

Step 4: Final Variance Formula

Finally, using the definition of variance, we subtract $(E[X])^2$ from $E[X^2]$ to get the variance formula:

$$\text{Var}(\sum_{i=1}^{T} X) = \sum_{i=1}^{B} P(x_i) \cdot (B - 2 \cdot b_i)^2 - (E[X])^2$$

3.2 calc_prob Function

The `calc_prob()` function takes the `entry_no` as input, replaces any 0's with 9's in the entry_no, and calculates the probability that Alice wins against Bob using the `CalculateProbability` class defined earlier.

3.3 Expectation Value

The `calc_expectation()` function calculates the expected value of the difference between Bob's wins and twice Alice's wins. It computes the probability for every possible value of Bob's wins and sums the products of the probability and the difference between the wins.

3.4 Variance Calculation

The `calc_variance()` function calculates the variance of the difference between Bob's wins and twice Alice's wins. The variance is calculated by first summing the square of the difference multiplied by the probability, and then subtracting the square of the expected value.

Solution for Question 2a: Greedy Srategy

The goal of this code is to perform Monte Carlo simulations to evaluate a game between two players, Alice and Bob, where Alice uses a "greedy" strategy based on her previous round results, while Bob follows a simpler, reactive strategy. The number of simulations is set to 10^5, and the outcomes of each round are determined by a payoff matrix.

4 Class Definitions

4.1 Class `GreedyAlice`

The class `GreedyAlice` inherits from `PlayerBase` and represents Alice's strategy. Her move selection is based on the result of the previous round. The key logic is implemented in the method `play_move`:

- If Alice won the last round (i.e., `self.results[-1] == 1`), she considers her own points and her opponent's points to decide her next move:
  - If $5 \times$ `opp_points` $> 6 \times$ `points`, Alice chooses move 0 (a aggressive strategy).

  - Otherwise, she chooses move 2 (an defensive strategy).

- If the last round was a draw (i.e., `self.results[-1] == 0.5`), Alice plays move 0.
- If Alice lost the last round (i.e., `self.results[-1] == 0`), she plays move 1.

  Where in play_move :

  - 0 denotes the aggressive strategy,
  - 1 denotes the balanced strategy,
  - 2 denotes the defensive strategy.

  This behavior aligns with the "greedy" strategy, where Alice tries to maximize her chances of winning based on the current state of the game and the last outcome.

4.2 Explanation:

When Alice wins the last round, Bob plays aggressively. Then, Alice has a choice of playing aggressively, balanced, or defensively. As we can see from the payoff matrix, the probability of Alice winning is less than the probability of Bob winning when Alice is playing balanced, so the choice is between aggressive and defensive strategies.

The aggressive strategy depends on n_A and n_B. By comparing the two, we observe the following:

The ratio of $P(\text{Alice Winning}) : P(\text{Bob Winning})$ for the defensive strategy is $6 : 5$,

the ratio of $P(\text{Alice Winning}) : P(\text{Bob Winning})$ for the aggressive strategy is $\frac{n_B}{n_A}$.

Thus, if $5 \times$ **Bob wins** $> 6 \times$ **Alice wins**, Alice should play aggressively. Otherwise, Alice should play defensively.

4.3 Class `Bob`

The class `Bob` also inherits from `PlayerBase` and represents Bob's simpler strategy. Bob's strategy is purely reactive and depends only on the result of the last round:

- If Bob won the last round (i.e., `self.results[-1] == 1`), he plays move 2 (Defensive).
- If the last round was a draw (i.e., `self.results[-1] == 0.5`), Bob plays move 1.
- If Bob lost the last round (i.e., `self.results[-1] == 0`), he plays move 0 (Aggresive).

This strategy is straightforward and does not take into account any other variables except the previous result.

4.4 Monte Carlo Simulation

The simulation is executed within the `if __name__ == "__main__":` block, which runs the `monte_carlo` function to simulate 10^5 rounds between Alice and Bob using their respective strategies. The function performs the following:

- Alice and Bob select their moves based on their respective strategies.
- The outcome of each round is determined using a payoff matrix (not shown here but assumed to be predefined).
- The game continues for multiple rounds, updating the players' scores and strategies after each round.
- After 10^5 rounds, the simulation prints the results, allowing for statistical analysis of the outcomes.

5 Solution for Question 2b

After analysis of the dynamic programming , I concluded that there is no non greedy analysis which is better than the greedy analysis, can be seen in the Codes.

6 Solution for Question 2c: Estimating $\mathbb{E}[\tau]$

In this solution, we define two players: **Alice** and **Bob**, with distinct strategies based on the outcomes of previous rounds. We aim to compute the expected number of rounds it takes for Alice to win a specific number of rounds.

Class Alice

The `Alice` class is derived from the base class `PlayerBase`. Alice starts with 1 win, and her strategy depends on the result of the previous round:

- If Alice won the last round (`results[-1] == 1`), she checks whether

  $$5 \times \texttt{opp\_points} \geq 6 \times \texttt{points}.$$

  If true, she plays move 0; otherwise, she plays move 2.
- If the previous round resulted in a tie (`results[-1] == 0.5`), Alice plays move 0.
- Otherwise, Alice plays move 1.

After each round, Alice updates her results and increments her win count if she wins the round.

Class Bob

The `Bob` class is also derived from the base class `PlayerBase`. Bob's strategy is:

- If Bob won the last round (`results[-1] == 1`), he plays move 2.
- If the previous round resulted in a tie (`results[-1] == 0.5`), Bob plays move 1.
- Otherwise, Bob plays move 0.

Expectation Calculation

The function `expectation_val` computes the expected number of rounds it takes for Alice to win a specific number of rounds. The input `entry_no` is processed to determine the target number of Alice's wins, denoted as t. The game is simulated 10^5 times, and in each simulation:

- Alice and Bob play rounds until Alice wins t rounds.
- The total number of rounds played (denoted as `tau`) is the sum of both players' points, i.e.,

$$\tau += \texttt{alice.points + alice.opp\_points}$$

The expected value of total rounds, $E[\tau]$, is calculated by averaging the total number of rounds over all simulations:

$$E[\tau] = \frac{\tau}{10^5}$$

where τ is the accumulated number of rounds over 10^5 simulations. This value is returned as the result.

7 Solution for Question 3:

In this solution, we define two players: **Alice** and **Bob**, where Bob plays a random move every time. We wish to find the optimal strategy to (a) maximise expected points for Alice in 1 round and (b) to maximise the expected points for Alice in a given number of rounds.

7.1 Solution for Question 3a

Class Alice

The `Alice` class is derived from the base class `PlayerBase`. Alice starts with 1 win, and her strategy for maximizing expectation value for 1 round depends purely on the number of points that Alice has and the number of points that Bob has at that instant:

- If the probability of Alice winning through attack is greater (**self.opp_points / (self.points + self.opp_points) + 127 / 110 $\geq$ 329 / 220**), she plays move 0; otherwise, she plays move 2.
- Playing balanced is not optimal ever, because of the subpar expectation value that it provides

Class Bob

The `Bob` class is also derived from the base class `PlayerBase`. Bob's strategy is:

- Bob plays a random move every turn, uninfluenced by other factors.

7.2 Solution for Question 3b

For this question, we work out the most optimal strategy for Alice through Dynamic Programming, through 2 dimensional dynamic programming, we can efficiently calculate the best strategy for Alice to have the maximum possible expectation value after T turns.

7.2.1 Dynamic Programming Logic

Here our 2 dimensional DP variable holds the following state: dp[i][j] = expected value of points that Alice will gain from this point on given that she has i/2 points and Bob has j/2 points to reach a total of num_rounds rounds.

We introduce this factor of 2 in order to have indexing for games which are drawed (2*0.5 = 1)

A top-down DP approach gives us the following recursion equation:

Through the DP matrix, we keep track of the played move at every point where `calc_dp` is executed in order to print the output for the strategy given the inputs n_a, n_b and total number of games.

$$\begin{aligned}\mathrm{dp}[a][b] = \max_{i\in\{0,1,2\}} [&(1+\mathrm{dp}[a][b+2]) \cdot P(\text{Alice wins given that she plays strategy } i) \\ &+ (0.5+\mathrm{dp}[a+1][b+1]) \cdot P(\text{Alice draws given that she plays strategy } i) \\ &+ (\mathrm{dp}[a][b+2]) \cdot P(\text{Alice loses given that she plays strategy } i)]\end{aligned}$$