# THE AUCTION BIDDING PROBLEM

- *SAIUDITI, EE22B139*

The auction being run for 3 variants, and 1000 rounds, with the previous 100 runs worth of data, and having the data of the highest_bidders and the second:highest_bidders and the winning values provides a very interesting case.

The structure of the report deals with general information related to the auction learnt during the research process, general pointers to keep in mind while creating bots during the auction, and then the models chosen for the three variants of the bidding problem presented, and the reason why they fare better than the other possible bots that could have been chosen.

***Takeaways relating to the problem statement,***

- The auction model adapted here is of the **first-price sealed auction** category.
- All the bidding values belong to the category of IPV, **Independent Private Values**.
- There is **no information asymmetry** among the players in the auction, that is the 20 bots, all bots have the same information, that is, relevant information of the previous 100 rounds.
- To not fall into the **Winner's Curse**, that happens as a result of overbidding, which eventually ends up in the winner winning the bid, but losing capital.
- The idea of **Bayes-Nash equilibrium**, and how all the bots are smart and there is not a way to make them smarter post the auction has started.
- The Expected Mean Payoff is a good way to go about things, and a risk function, complex as it maybe could be included that could be minimised.
- Game theories can be applied to auction strategies.
- Approaches that were researched and explored before concluding to the ones chosen include,
  - Deep Q Network
  - Monte Carlo Tree Search
  - Epsilon Greedy Method
  - Upper Confidence Bound
  - Evolutionary Algorithm
  - Particle Filter
  - Hidden Markov Model
  - Deep Reinforcement learning
  - Thomson Sampling
  - Markov Decision Process
  - KNN
  - Extreme Gradient Boosting (XGBoost)
  - Random Forest Classifiers

- K Means Clustering
- Stackelberg Competition
- Moving average with Weighted Exponential Smoothing
- Reinforcement Learning
- Dynamic Threshold Bidding and Regret Minimisation Strats
- Linear and Polynomial Regression
- Implementing a neural network, through which outputs of models like Regression and Random Forest Classifiers were tried to be passed
- Nash Equilibrium Bidding
- Multi-Armed Bandits

(credits to AI - ChatGPT, for helping write the code for a lot of the newer methods)

***General Takeaways:***

- *Capital and Risk Management:* Overbidding might lead to losses, or the term 'Winner's curse'.
- *Bidding Logic:* Data of the past 100 rounds would help to figure out 'some' patterns.
- *Understanding patterns of Opponents/Rounds and an Adaptive Strategy*
- *Initial Rounds' Exploratory Phase, Mid-game's Exploitation, (and maybe an endgame aggressive bidding phase to maximise payoff) Phase for Variation 2 and 3*
- *Capital Threshold Strategy:* We ensure to not overly bid, while the 'x_i's assigned to our bot is low.
- *For Variation 3:* Avoiding the second-highest position at all costs. This can be done by learning the gap between the highest and second-highest bids over the last 100 rounds and adjusting the bid accordingly. Either aiming for first place or deliberately underbidding to avoid losing capital.

---

# VARIATION ONE

**Variation One Code:**

```
from Strategy import StrategyBase
import numpy as np
import random
from Strategy import StrategyBase  # Assuming StrategyBase is defined in your setup

class UserStrategy(StrategyBase):
    def __init__(self):
        # Correctly initialize attributes in the constructor
        self.previous_bids = []  # Track past bids for analysis
        self.previous_winners = []
        self.previous_second_highest_bids = []

    def make_bid(self, current_value, previous_winners, previous_second_highest_bids, capital, num_bidders):
        '''
        Optimized bidding strategy based on Expected Value, Adaptive Bidding, and Capital Management.
        '''
        # Bayes-Nash Equilibrium Strategy: b(x_i) = (n-1)/n * x_i
        bne_bid = (num_bidders - 1) / num_bidders * current_value

        # Default bid based on adaptive learning and risk management
        if previous_winners and previous_second_highest_bids:
            avg_winner = np.mean(previous_winners[-10:])  # Use last 10 rounds
            avg_second_highest = np.mean(previous_second_highest_bids[-10:])
        else:
            avg_winner = current_value * 0.75
            avg_second_highest = current_value * 0.5

        # Adaptive Bid: Based on current value, historical data, and randomness
        adaptive_bid = min(current_value * 0.8, avg_winner + random.uniform(-0.05, 0.05) * current_value)

        # Final calculated bid, balancing between Bayes-Nash and adaptive strategy
        base_bid = max(bne_bid, adaptive_bid)

        # Capital Management: Bid only a portion of your capital to stay safe
        safe_bid = min(base_bid, capital * 0.25)

        # Introduce randomness to avoid predictable patterns
        random_factor = random.uniform(0.97, 1.03)
        final_bid = safe_bid * random_factor

        # Capital constraints: Don't bid more than your capital allows
        final_bid = min(final_bid, capital)

        # Keep track of previous bids, winners, and second-highest bids
        self.previous_bids.append(final_bid)
        self.previous_winners = previous_winners
        self.previous_second_highest_bids = previous_second_highest_bids

        return final_bid
```

## Variation One approach adapted:

## The strategy adapted uses a combination of theoretical approaches to bidding,

- **Bayes-Nash concept:**

This ensures the bot does not overbid, but still is competitive enough, and changed its bid values according to the number of players remaining in the game. The bid is based on a sound game-theoretic principle.

$$b(x_i) = \frac{(n-1)}{n} \times x_i$$

wherein ‘n’ is the number of bidders’ x_i’ is the bot’s bid value

- **Adaptive Learning:**

This allows the bot to adapt to changing patterns in the game, and use historical data of the previous 100 rounds. This helps the bot adjust to the behavior of other bidders and stay competitive, and adapt itself to the dynamic real world auction bots.

- **Capital Management aspect:**

The bot limits the bid to 30% of the remaining capital, maintaining a buffer for future rounds, and preventing overbidding which could lead to losses.

- **Randomness Introduction:**

A random factor is introduced to the final bid to avoid predictability. The bid is scaled by a random factor between 0.95 and 1.05, to reduce chances of being the bot's theory to be predicted by other models, although within an auction of 20 bots, this does not make much difference. With dwindling number of bots, this will work better.

- **Tracking and Learning**:

The previous winners, and second highest winners are collected and the bot learns and tries to see patterns to exploit.

***Why the model adopted works better than other models, like Linear Regression, Reinforcement Learning, or XGBoost, or Machine Learning approaches?***

- By incorporating *capital management and randomness*, the strategy *minimizes risks and avoids predictable patterns*. This can be advantageous in competitive environments where predictability can be exploited by opponents.
- The strategy blends *theoretical concepts with practical adjustments*. This hybrid approach leverages the strengths of both *game theory and adaptive learning*.
- Sometimes Machine Learning models might tend to overfit data, and this approach prevents that from happening.

---

# VARIATION TWO

### Variation Two Code:

```
class CombinedStrategy(StrategyBase):
    def __init__(self):
        # Polynomial Regression model setup
        self.poly_features = PolynomialFeatures(degree=3)  # Increased degree for more accuracy
        self.model = make_pipeline(self.poly_features, LinearRegression())
        self.epsilon = 0.1  # Exploration parameter for MAB
        self.trained = False

    def train_model(self, previous_winners, previous_second_highest_bids):
        # Ensure there are enough data points for training
        if len(previous_winners) < 10:
            return
        # Train Polynomial Regression model
        X = np.array(previous_second_highest_bids[-10:]).reshape(-1, 1)
        y = np.array(previous_winners[-10:])
        self.model.fit(X, y)
        self.trained = True

    def make_bid(self, current_value, previous_winners, previous_second_highest_bids, capital, num_bidders):
        if len(previous_winners) < 3:
            return min(capital, 10)  # Default bid if not enough history

        # Step 1: Train the model if enough data is available
        if not self.trained:
            self.train_model(previous_winners, previous_second_highest_bids)

        # Step 2: Polynomial Regression Prediction
        if self.trained:
            predicted_winner = self.model.predict([[previous_second_highest_bids[-1]]])[0]
        else:
            predicted_winner = np.mean(previous_winners)  # Fallback if not enough data

        # Step 3: Moving Average for Trend Stability
        moving_avg = np.mean(previous_winners[-5:])

        # Step 4: Combine both predictions (Polynomial Regression and Moving Average)
        combined_bid = (predicted_winner + moving_avg) / 2

        # Step 5: Multi-Armed Bandit Exploration (explore vs exploit)
        if random.uniform(0, 1) < self.epsilon:
            # Exploration: Random bid within a reasonable range based on capital
            bid = random.uniform(0, min(capital, current_value))
        else:
            # Exploitation: Use the combined prediction adjusted by capital
            safe_bid = min(combined_bid - 5, capital * 0.4)  # Capital-aware adjustment
            bid = max(safe_bid, 1)  # Ensure we don't bid too low

        # Step 6: Final bid with capital constraint
        bid = min(bid, capital)

        return bid
```

### Variation Two Strategy adapted:

- The strategy here is a combination of strategies involving Polynomial Regression, Moving Averages, and a Multi-Armed Bandit (MAB) approach, to make strategic bids in the auction game.

- **Polynomial Regression:**

The technique of Polynomial Regression has been used to *predict the next winning bid* based on the history of the 100 rounds that have been run and the rounds that are run post that. A third degree polynomial model has been used, to be able to figure the *trends between the highest and the second highest bids*. This helps identify hidden patterns in how bids fluctuate, accounting for factors like *bidder behavior, capital limits, and competition* over time.

The second-highest bid is crucial because it is a reflection of how aggressive other bidders are.directly influences the dynamics of the auction. Determining the second highest bid, helps to make an optimal winning bid prediction without overbidding, and thereby helps maximising payoff.

- **Moving Averages:**

The strategy incorporates a moving average of the winning bids over the last 5 rounds, to stabilize the predictions and reduce volatility. The Moving Average captures recent trends and *smooths out fluctuations* that might arise from *outliers*, and ensures that the strategy does not overreact to temporary changes in the auction dynamics. The moving average helps ensure that the strategy doesn't. The moving average helps ensure that the *strategy doesn't overreact* to *temporary changes in the auction dynamics*.

- **Combination of Polynomial Regression and Moving Averages:**

The moving average is combined with the Polynomial Regression prediction to create a more balanced bid. While polynomial regression captures patterns and predicts the next winning bid based on past second-highest bids, the moving average gives a steady indication of recent winning bid trends.his creates a hybrid prediction that balances the *precision of polynomial regression with the stability of the moving average*, thus combining aspects of short term and long term prediction.

- **Multi-Armed Bandits (MABs):**

Multi Armed Bandit helps in balancing *exploration* that is, trying new strategies, and *exploitation*, that is, using learned strategies. It also introduces with a small probability ($\varepsilon = 0.1$ *or 10%*), a random bid with a reasonable range to allow it to explore potentially viable bidding strategies. It uses a combination of predictions obtained as a result of *performing polynomial regression and moving averages.*

The *exploration parameter* at  0.1, ensures that the strategy will randomly explore a bid (10% of the time). This is useful for discovering new opportunities that may not be obvious from the historical data. During exploration, the bid is a random value between 0 and the minimum of the player's available capital and current value.

During exploitation (90% of the time), the strategy relies on the predictions from Polynomial Regression and the moving average.

The *epsilon-greedy approach is well-suited for balancing experimentation with known strategies in dynamic environments*, like this bidding game. The exploration phase allows the strategy to stay flexible and adaptive, while exploitation ensures it's mostly relying on the informed predictions based on past data.

- **Capital Constraints:**

The strategy ensures to prevent reckless bidding and ensures bidding is capped at 40% of the available capital. To ensure participation in rounds, a bid of 1 has been set as the minimum bid.

***Why the model adopted works better than other models, like Linear Regression, Reinforcement Learning, or XGBoost, or Machine Learning approaches?***

- By combining predictions from polynomial regression with the moving average, the strategy benefits from *both sophisticated modeling and trend smoothing*.
- This is better than linear regression, as *linear regression might not be able to capture the non-linear relationship* between the highest and second highest bid.
- The Multi-Armed Bandit strategy ensures to *not become overly reliant on a specific prediction model based on machine learning*, and can take an explorative-exploitative approach to adapt to changing dynamics.
- There *exists is a fallback mechanism*, such that even if there is insufficient data and training in machine learning models cannot be performed, this strategy resorts to using an average of the previous winning bids. Even with limited data or unpredictably large variations, the model does fairly well.
- There is a *dynamic adaptation*, and the bot adapts based on the outcomes and thus can survive in extremely competitive environments wherein the other bots' strategies also evolve over time.
- The robustness is also exemplified by the fact that the approach can provide a more balanced and robust bid prediction compared to relying solely on one technique.

---

# VARIATION THREE

## Variation Three Code:

```
from Strategy import StrategyBase
import numpy as np
import xgboost as xgb

class UserStrategy(StrategyBase):

    def __init__(self):
        # Initialize the XGBoost model
        self.model = xgb.XGBRegressor(n_estimators=200, max_depth=6, learning_rate=0.05, random_state=42)
        self.is_trained = False
        self.previous_winners = []
        self.previous_second_highest_bids = []
        self.safety_margin = 0.02  # Ensure profit margin
        self.aggressiveness = 0.1  # Adjust aggressiveness for bidding

    def update_data(self, winner, second_highest_bid):
        """Update historical data with new round results."""
        self.previous_winners.append(winner)
        self.previous_second_highest_bids.append(second_highest_bid)

        # Keep only the last 100 rounds of data
        if len(self.previous_winners) > 100:
            self.previous_winners.pop(0)
            self.previous_second_highest_bids.pop(0)

    def train_model(self):
        """Train the XGBoost model if there is enough data."""
        if len(self.previous_winners) >= 20:  # Require at least 20 rounds of data
            X_train = np.array(self.previous_winners).reshape(-1, 1)
            y_train = np.array(self.previous_second_highest_bids)

            # Train the model
            self.model.fit(X_train, y_train)
            self.is_trained = True

    def make_bid(self, current_value, previous_winners, previous_second_highest_bids, capital, num_bidders):
        '''
        This function makes a bid for the auction:
        1. Predicts the second-highest bid using XGBoost.
        2. Ensures that the bid allows for profit while keeping capital in mind.
        3. The strategy minimizes risk for both winner and second-highest bidder.
        '''
        # Update data with latest round results
        if previous_winners and previous_second_highest_bids:
            self.update_data(previous_winners[-1], previous_second_highest_bids[-1])

        # Train the model
        self.train_model()

        # Step 1: Predict second-highest bid using XGBoost
        if self.is_trained:
            predicted_second_highest = self.model.predict(np.array([[current_value]]))[0]
        else:
```

```
            # Fallback if not enough data to train the model
            predicted_second_highest = current_value * 0.75

        # Step 2: Adjust bid based on predicted second-highest bid and ensure profit
        if capital > current_value * 0.5:
            # More aggressive if capital is high: bid above second-highest prediction
            target_bid = predicted_second_highest * (1 + self.aggressiveness)
        else:
            # Conservative bidding if capital is low
            target_bid = predicted_second_highest * (1 - self.safety_margin)

        # Step 3: Ensure positive profit for the winner
        bid = min(target_bid, capital)  # Bid must not exceed available capital
        bid = max(0, bid)  # Ensure bid is not negative
        bid = min(bid, current_value)  # Ensure bid is not more than player's value

        # Step 4: Adjust to avoid negative profit for second-highest bidder
        if current_value - bid <= 0:
            bid = current_value * (1 - self.safety_margin)

        return bid
```

## Variation Three approach adapted:

The strategy adapted uses *Extreme Gradient Boosting (XGBoost)* for this variation,

XGBoost generally works well with 20 rounds of data atleast, it uses a *training data*, which consists of an array of winners and second highest bids. The trained model is used to predict the second highest bid, and incase the model is not trained it defaults to 75% if current_value as a fallback.

If the bot has capital more than 50% of current value, it bids aggressively more than the second highest bid by a factor of self.aggressiveness, that is a 10% more. If the capital is less, then the bid is adjusted conservatively by reducing the predicted bid by self.safety margin, that is a 2% to ensure a profit margin.

***Why the adopted XGBoost works better than other models, like Linear Regression, Reinforcement Learning, or a combination of mathematical approaches?***

- The model is able to handle *non linearity, and complexities*, and especially with the new constraint, or never being the second highest bidder, the XGBoost, builds an ensemble of decision trees iteratively, wherein each tree corrects the errors made by the previous tree, proving particularly useful in scenarios like this wherein the relation between the function parameters, and the target variable, which is the payoff, happens to be non-linear in nature.
- The *interactions between the highest and second highest*, with wonderfully intelligent bots might prove to be a challenge to capture, The XGBoost does way better in capturing them than linear models or simpler linear machine learning techniques.
- XGBoost is able to incorporate *regularization terms*, specifically the L1 and L2 regularization and *prevents overfitting* thus, which is crucial in this bidding auction variant. It also has built-in early stopping capabilities, which allows it to stop training when additional boosting is not able to improve performance much.
- XGBoost takes *feature importance scores* into account, which helps weighing features that contribute most to the payoff.
- XGBoost is able to deal with *custom loss functions*, and the bidding problem can have a tailored loss function for the payoff and penalties for the second highest bidder.
- The decision tree-based structure of XGBoost can also handle situations with *outliers* being present in the data (and missing input values). XGBoost is capable of parallel processing, and effectively handles *training time and speed of running*.

*Musings of Runs of 40ish bots running against each other, and minutes of runtime, with yes very funny names,  XD, with returns varying from 7k to 1k in 15 to 40 bot rounds respectively:*

Variation3:

```
Top 2 bids are 99.65,93.07
c2: bid - 50.01, initial_capital - 500.00, value - 99.00, capital left - 500.00
c3: bid - 50.01, initial_capital - 500.00, value - 75.00, capital left - 500.00
curr trying: bid - 58.47, initial_capital - 550.16, value - 60.00, capital left - 550.16
curr: bid - 50.01, initial_capital - 500.00, value - 99.00, capital left - 500.00
e2: bid - 71.38, initial_capital - 510.76, value - 66.00, capital left - 510.76
e3: bid - 2.82, initial_capital - 529.69, value - 3.00, capital left - 529.69
```

```
epsilon: bid - 0.00, initial_capital - 509.22, value - 53.00, capital left - 509.22
esteemed: bid - 16.13, initial_capital - 502.55, value - 9.00, capital left - 502.55
hey: bid - 82.12, initial_capital - 82.12, value - 35.00, capital left - 82.12
heyyyyyyy: bid - 82.12, initial_capital - 82.12, value - 14.00, capital left - 82.12
hiii: bid - 82.12, initial_capital - 82.12, value - 39.00, capital left - 82.12
lastensemble: bid - 26.60, initial_capital - 551.30, value - 28.00, capital left - 551.30
lastRFOREST: bid - 20.79, initial_capital - 592.40, value - 21.00, capital left - 592.40
lastxg_var3333: bid - 17.64, initial_capital - 546.15, value - 18.00, capital left - 546.15
mock1: bid - 43.00, initial_capital - 558.86, value - 26.00, capital left - 558.86
mock2: bid - 99.65, initial_capital - 541.12, value - 87.00, capital left - 540.47
montecarlo: bid - -1.00, initial_capital - 500.00, value - 95.00, capital left - 500.00
new: bid - 16.13, initial_capital - 500.00, value - 37.00, capital left - 500.00
new2: bid - 40.48, initial_capital - 532.75, value - 43.00, capital left - 532.75
particlefilter: bid - 0.72, initial_capital - 500.00, value - 90.00, capital left - 500.00
rando: bid - 89.09, initial_capital - 480.15, value - 16.00, capital left - 480.15
randotryinggggg: bid - 90.45, initial_capital - 607.57, value - 52.00, capital left - 607.57
run: bid - 70.99, initial_capital - 524.76, value - 75.00, capital left - 524.76
stackelberg: bid - -2.00, initial_capital - 494.51, value - 9.00, capital left - 494.51
Test Strategy 1: bid - 0.00, initial_capital - 503.01, value - 55.00, capital left - 503.01
Test Strategy 1: bid - -1.00, initial_capital - 500.00, value - 45.00, capital left - 500.00
Test Strategy 2: bid - -2.00, initial_capital - 508.70, value - 17.00, capital left - 508.70
Test Strategy 3: bid - 10.00, initial_capital - 500.00, value - 5.00, capital left - 500.00
tryying: bid - 90.30, initial_capital - 607.63, value - 91.00, capital left - 607.63
tryyyy: bid - 41.61, initial_capital - 573.34, value - 45.00, capital left - 573.34
UCB: bid - 0.00, initial_capital - 500.00, value - 85.00, capital left - 500.00
Var1_EE2B139_2: bid - 6.07, initial_capital - 544.90, value - 6.00, capital left - 544.90
var2MABNIGHT!: bid - 0.00, initial_capital - 500.00, value - 17.00, capital left - 500.00
var2try: bid - 93.07, initial_capital - 1220.77, value - 23.00, capital left - 1220.77
Var2_EE22B139: bid - 85.76, initial_capital - 486.17, value - 93.00, capital left - 486.17
var3eve: bid - -2.00, initial_capital - 499.00, value - 72.00, capital left - 499.00
```

Variation2:

```
epsilon: bid - 0.00, initial_capital - 500.00, value - 64.00, capital left - 500.00
esteemed: bid - 28.40, initial_capital - 505.87, value - 30.00, capital left - 505.87
hey: bid - 97.00, initial_capital - 97.00, value - 12.00, capital left - 97.00
heyyyyyyy: bid - 97.00, initial_capital - 97.00, value - 51.00, capital left - 97.00
hiii: bid - 97.00, initial_capital - 97.00, value - 4.00, capital left - 97.00
lastensemble: bid - 89.30, initial_capital - 563.00, value - 94.00, capital left - 563.00
lastRFOREST: bid - 34.65, initial_capital - 609.69, value - 35.00, capital left - 609.69
lastxg_var3333: bid - 38.22, initial_capital - 581.83, value - 39.00, capital left - 581.83
mock1: bid - 93.00, initial_capital - 571.05, value - 14.00, capital left - 571.05
mock2: bid - 57.58, initial_capital - 521.69, value - 65.00, capital left - 521.69
montecarlo: bid - -1.00, initial_capital - 500.00, value - 23.00, capital left - 500.00
new: bid - 27.60, initial_capital - 500.00, value - 39.00, capital left - 500.00
new2: bid - 84.32, initial_capital - 508.04, value - 91.00, capital left - 508.04
rando: bid - 55.25, initial_capital - 595.32, value - 82.00, capital left - 595.32
randotryinggggg: bid - 93.16, initial_capital - 1109.10, value - 26.00, capital left - 1109.10
run: bid - 0.00, initial_capital - 543.93, value - 0.00, capital left - 543.93
stackelberg: bid - -2.00, initial_capital - 502.21, value - 76.00, capital left - 502.21
Test Strategy 1: bid - 0.00, initial_capital - 519.64, value - 23.00, capital left - 519.64
Test Strategy 1: bid - -1.00, initial_capital - 500.00, value - 58.00, capital left - 500.00
Test Strategy 2: bid - -2.00, initial_capital - 505.21, value - 100.00, capital left - 505.21
Test Strategy 3: bid - 10.00, initial_capital - 500.00, value - 20.00, capital left - 500.00
tryying: bid - 12.04, initial_capital - 559.60, value - 12.00, capital left - 559.60
tryyyy: bid - 93.88, initial_capital - 536.07, value - 98.00, capital left - 536.07
UCB: bid - 0.00, initial_capital - 500.00, value - 40.00, capital left - 500.00
Var1_EE2B139_2: bid - 62.98, initial_capital - 559.61, value - 64.00, capital left - 559.61
```