Project 1: Building a Second-Price Auction (Modeling and Strategy)

You has been hired by a major retailer to develop algorithms for an online ad auction. Your client knows a little about the multi-armed bandit literature and recognizes that it can spend money to explore, learning how likely users are to click on ads, or to exploit, spending on the most promising users to maximize immediate payoffs. At the same time, there are other companies participating in the auction that may outbid your client, potentially interfering with these goals. Your task is to model the ad auction and develop an effective algorithm for bidding in a landscape of strategic competitors. Your client plans to test your bidding algorithm against other bidding algorithms contributed by other data scientists, in order to select the most promising algorithm.

The Auction Rules

The `Auction` is a game, involving a set of `Bidder`s on one side, and a set of `User`s on the other. Each round represents an event in which a `User` navigates to a website with a space for an ad. When this happens, the `Bidder`s will place bids, and the winner gets to show their ad to the `User`. The `User` may click on the ad, or not click, and the winning `Bidder` gets to observe the `User`'s behavior. This is a second price sealed-bid `Auction`.

There are `num_users` `User`s, numbered from 0 to `num_users - 1`. The number corresponding to a user will be called its `user_id`. Each user has a secret probability of clicking, whenever it is shown an ad. The probability is the same, no matter which `Bidder` gets to show the ad, and the probability never changes. The events of clicking on each ad are mutually independent. When a user is created, the secret probability is drawn from a uniform distribution from 0 to 1.

There is a set of `Bidder`s. Each `Bidder` begins with a balance of 0 dollars. The objective is to finish the game with as high a balance as possible. At some points during the game, the `Bidder`'s balance may become negative. If you `Bidder`'s balance goes below -1000 dollars then your `Bidder` will be disqualified from the `Auction` and further bidding.

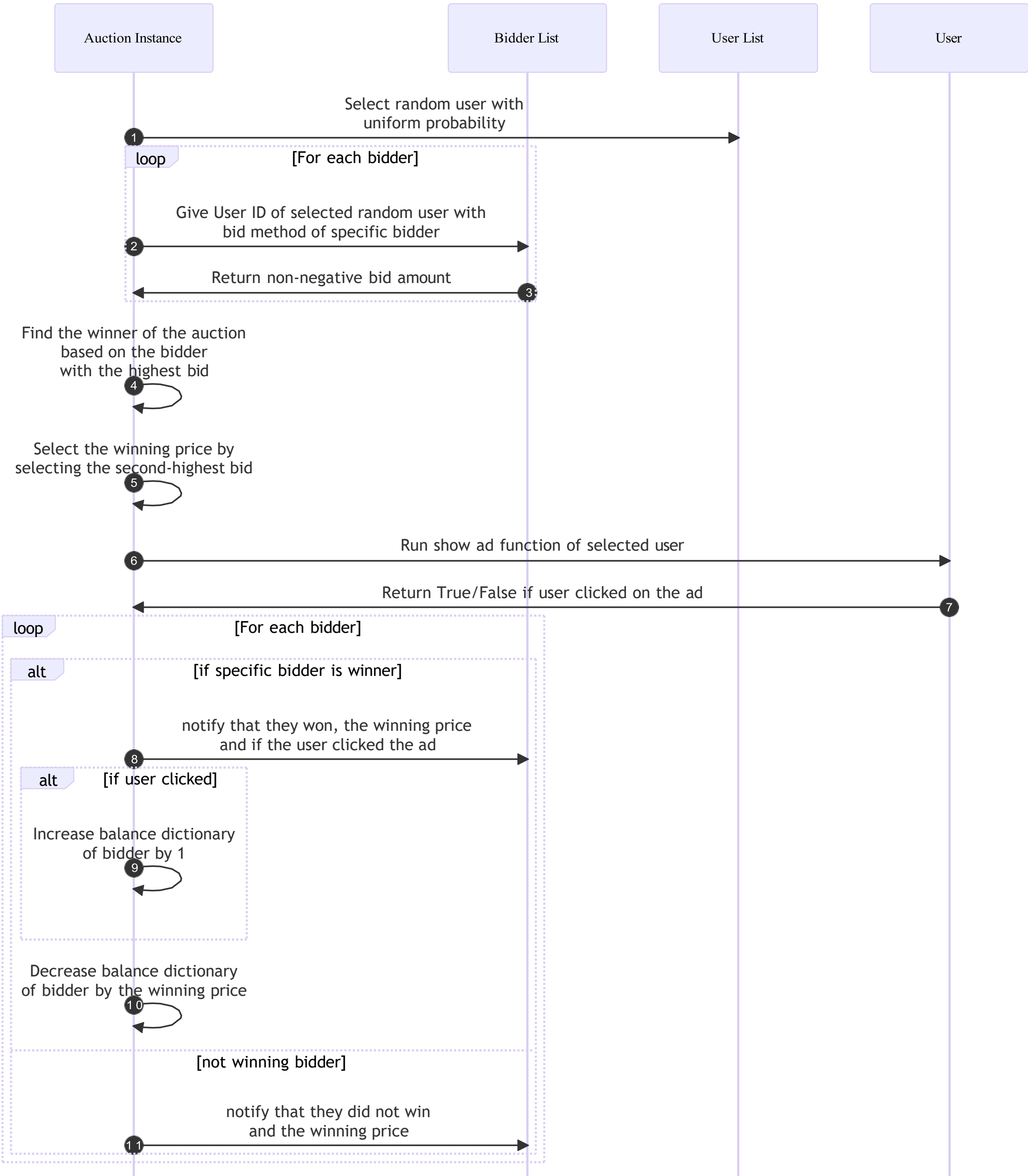
The `Auction` occurs in rounds, and the total number of rounds is `num_rounds`. In each round, a second-price auction is conducted for a randomly chosen `User`. Each round proceeds as follows:

1. A `User` is chosen at random, with all `User`s having the same probability of being chosen. Note that a `User` may be chosen during more than one round.
2. Each `Bidder` is told the `user_id` of the chosen `User` and gets to make a bid. The bid can be any non-negative amount of money in dollars. A `Bidder` does not get to know how much any other `Bidder` has bid.
3. The winner of the auction is the `Bidder` with the highest bid. In the event that more than one `Bidder` ties for the highest bid, one of the highest `Bidder`s is selected at random, each with equal probability.
4. The winning price is the second-highest bid, meaning the maximum bid, after the winner's bid is removed, from the set of all bids. If the maximum bid was submitted by more than one bidder then the second price will be the maximum bid. For example, if two bidders bid 2 and no one else bids higher then 2 is the winning price.
5. The `User` is shown an ad and clicks or doesn't click according to its secret probability.
6. Each `Bidder` is notified about whether they won the round or not, and what the winning price is. Additionally, the winning `Bidder` (but no other `Bidder`) is notified about whether

the `User` clicked.

7. The balance of the winning `Bidder` is increased by 1 dollar if the `User` clicked (0 dollars if the user did not click). It is also decreased by the winning price (whether or not the `User` clicked).

Sequence Diagram: Execute Round



Architecture

You are asked to design the following classes:

A `User` class that includes:

- an initializer method with the definition `def __init__(self)` .
- a private `__probability` attribute to represent the probability of clicking on an ad. When a user is created, the secret probability is drawn from a uniform distribution from 0 to 1. (Please use the random or numpy.random modules)
- a `show_ad` method with the definition `def show_ad(self)` that represents showing an ad to this `User`. This method should return `True` to represent the user clicking on and ad and `False` otherwise.

A `Bidder` class that includes:

- an initializer with the definition `def __init__(self, num_users, num_rounds)`, in which `num_users` contains the number of `User` objects in the game, and `num_rounds` contains the total number of rounds to be played. The `Bidder` might want to use this info to help plan its strategy.
- a `bid` method with the definition `def bid(self, user_id)`, which returns a non-negative amount of money, in dollars round to three (3) decimal places.
- a `notify` method with the definition `def notify(self, auction_winner, price, clicked)` , which is used to send information about what happened in a round back to the `Bidder`. Here, `auction_winner` is a boolean to represent whether the given `Bidder` won the auction (`True`) or not (`False`). `price` is the amount of the second bid, which the winner pays. If the given `Bidder` won the auction, `clicked` will contain a boolean value to represent whether the `user` clicked on the ad. If the given `Bidder` did not win the auction, `clicked` will always contain `None`.

An `Auction` class that includes:

- an initializer with the definition `def __init__(self, users, bidders)`. Here, `users` is expected to contain a list of all `User` objects. `bidders` is expected to contain a list of all `Bidder` objects.
- an `execute_round` method with the header `def execute_round(self)`. This method should execute all steps within a single round of the game.

- a `balances` attribute, which contains a dictionary of the current balance of every `Bidder`.
- (optional) a `plot_history` method with the definition `def plot_history(self)`, which creates a visual representation of how the auction has proceeded. It is up to you do decide what the graphic looks like, and this method is meant to help you assess how your algorithm is performing. matplotlib is covered in Module 12. **There is a problem with the autograder that it cannot import matplotlib so please comment this out before submitting**

Note: You can use the python standard library, numpy, matplotlib or seaborn only.

Warning: You may create additional attributes beyond those described above. However, during the competition and in testing your code, your classes will interact with those written by the instructors, so they can **only** interact through the methods listed above. For example, if you write your own `Auction.get_user` method and call it from your `Bidder`, that will cause an error when we test your `Bidder` against the instructor `Auction`.

Deliverable

You should submit two files both on gradescope and to your github repo. We will be pulling your github repo for this assignment so please ensure its up to date!

Titled `auction_lastname.py` and `bidder_lastname.py`:

- `auction_lastname.py` should include the class definition of `Auction` and the class definition of `User` and nothing else.
- `bidder_lastname.py` should include the class definition of `Bidder` and nothing else.

Your submission cannot include any other statement that are outside the classes (for example, print statements or statements to instantiate or test your classes.)
Do not include debugging print statements in your code! Please also delete any commented out code before your final submission.

Competition

Once all submissions are collected, your instructors will pit your submission against all other submissions in a final competition. In a competition game, one `Bidder` from each submissions will be created along with a set of `User`s. Points will be awarded based on the final ranking.

To make the competition more stable, the instructors will create a large number of games, and then average the point totals together. Games may vary in `num_users` and `num_rounds`.

It is highly suggested that you make a couple of different `Bidder`s in separate files and run your own 'mock' `Auction` to see how each bidding strategy performs.

Grading Rubric

The grading policy is designed to emphasize the foundations of object oriented coding, with a small percentage reserved for students to extend themselves beyond the basics in designing strategic algorithms.

- 80% - Correct implementation of the methods listed above.
- 20% - Readability and proper commenting of your code.
    - Use pep8 for coding standards: https://www.python.org/dev/peps/pep-0008/
    - Please ensure your classes & methods have docstrings
    - Code is well-commented

- We will use pylint (https://pylint.pycqa.org/en/latest/) to grade your code. You need above a score of 8 on both the auction.py and the bidder.py to get full points.
- You can download the library and run pylint and the command line to see your score.
- If you use an IDE, you can download pylint as a plugin also to see the score / what lines to fix.

- 5% - Extra credit for your performance in the competition against all other submissions across the class. The highest scoring students will earn a full 5 percentage points, dropping to 0 for the bottom scoring students.

Helpful Links:

If you would like to read more high-level background on a second-price auction try these links:

- https://smartyads.com/blog/smartyads-dual-auction-soft-transition-towards-first-price-auctions/
- https://voluumdsp.com/blog/what-is-the-difference-first-price-vs-second-price/