The Trading Agent Competition (TAC) is an annual tournament in which teams of programmers from around the world build autonomous agents that trade in a market simulation. Application domains for these market simulations have ranged from travel to supply chain management to ad auctions to energy markets, and most recently, to ad exchanges.

In this lab, you will build an agent to play a greatly simplified version of the most recent game, Ad Exchange (AdX), in which agents play the role of ad networks, competing to procure contracts (i.e., advertising campaigns) from advertisers (e.g., retailers), and then bidding in an exchange on opportunities to exhibit those ads to Internet users as they browse the web. The game is complicated by the fact that certain demographics are more valuable to certain retailers, but user demographics are not always fully visible to the ad networks. Moreover, bidding repeats over a series of simulated days (60), and agents’ reputations for successfully fulfilling past contracts impact their ability to procure future contracts.

Game Description

In this lab, you will build an agent to play a very simple version of the TAC AdX game. The primary simplifications are: your agent will not compete to procure campaigns (instead your agent will be assigned one, randomly, from a known distribution), user demographics are fully visible, and the game lasts only one day, so there are no reputation effects. Consequently, your agent’s only job in this game is to bid on (and win) impression opportunities, which are opportunities to exhibit ads to Internet users as they browse the web. And your goal will be for your agent to do so in such a way as to fulfill its campaign, as inexpensively as possible.

More specifically, the private information that your agent is given at the start of this game is a single campaign, which represents an advertiser coming to you and saying “This contract is for some number of ads to be shown to users in some demographic, and here is my budget.” That is, each campaign is characterized by the following:

- A market segment: a demographic(s) to be targeted.
  - There are 26 market segments in this game, corresponding to combinations of \{Male, Female\} \times \{HighIncome, LowIncome\} \times \{Old,
Young}. A market segment might target only one of these attributes (for example, only Female) or two (Female_Young) or three (Female_Old_HighIncome).

- **A reach**: the number of ads to be shown to users in the relevant market segment.
- **A budget**: the amount of money the advertiser will pay if this contract is fulfilled.

N.B. Campaigns also have start and end days. However, in the version we are playing in lab today, the entire game only lasts 1 day, so you can ignore start and end days.

In each simulation, there will be a certain number of Internet users browsing the web, with their market segments drawn from the distribution described at the end of this document. For each user, a second price sealed-bid auction is run to determine who to allocate that user’s advertising space to, and how much to charge. Ties are broken randomly per user. So if there are two winning bidders in a market segment, each will be allocated (about) half the users in that segment at the price they bid.

Here is an example of a campaign:
[Segment = Female_Old, Reach = 500, Budget = $40.0]

To fulfill this campaign, your agent must show at least 500 advertisements to older women. If successful, it will earn $40. Showing an advertisement to a user is equivalent to winning that user’s auction. But note that winning an auction for a user who does not match a campaign’s market segment does not count toward fulfilling that campaign.

At the end of each simulation, the server computes the profit earned by each agent/ad network. Profit is computed as the product of the proportion of the campaign’s reach fulfilled and the campaign’s budget, less total spending. The proportion of reach fulfilled only counts users won in the relevant market segment, and cannot be higher than 1 (it does not help an agent to win more users than its reach). The agent with the highest profit wins that simulation. Because of the randomness in each simulation, the game is simulated repeatedly and scores are averaged over multiple simulations to determine an overall winner.

**A Minor Complication**

Unlike the sealed-bid auctions we have studied in class, which are one-shot auctions (i.e., they happen only once, just like the games we studied at the start of the semester -- Prisoners’ Dilemma, etc.), the user auctions in this game are repeated. However, agents can only bid in these auctions once! -- before their simulation begins.
Consequently, agents must reason in advance about how events might unfold over the course of the day, and perhaps make contingency plans. The AdX game provides a mechanism for making a contingency plan in the form of a spending limit: i.e., an upper bound on spending that can accompany an agent’s bid in each market segment.

If your agent is allocated a campaign whose market segment is very specific (e.g., Female_Old_HighIncome), then it won’t have a choice about which users to bid on; it has to bid for users in precisely that market segment, or it cannot earn a positive profit. However, if its market segment is less specific (e.g., Female), it can bid different amounts in the Female_Old and Female_Young markets, for example, based on how much competition it thinks there will be in each. Keep in mind, though, that the order in which users arrive is random. So if it bids more on Female_Old than Female_Young, but then if all Female_Old users arrive before any Female_Young, it may end up spending its entire budget for that campaign on Female_Old users. For this reason, when bidding on a market segment, your agent might want to specify a spending limit specifically for that segment.

An agent can also specify an overall spending limit to ensure that it does not spend more than some fixed amount on all users. This feature may or may not be useful, depending on your strategy.

**API for AdX One-Day Game**

**OneDayBidBundle Object**

To avoid the gruesome communication overhead required to conduct each ad auction in real time (each day there are 10,000 simulated users!), the agents use a OneDayBidBundle object to communicate their bids to the server all at once.

The constructor for this OneDayBidBundle object takes 3 parameters:

1. **Campaign ID**: the ID for the campaign you are bidding on.
2. **Day Limit**: a limit on how much you want to spend in total on that day.
3. **Bid Entries**: a collection of SimpleBidEntry objects, which specify how much to bid in each market segment.

A SimpleBidEntry object has 3 parameters:

1. **Market Segment**: there are a total of 26 possible market segments.
2. **Bid**: a double value.
3. **Spending Limit**: a double value that represents the maximum value the agent is willing to spend in this market segment.
For example, say your agent decides to bid 1.0 in all market segments in your campaign, and it wants to limit spending in total and in each market segment to the campaign's budget. First it would create a new SimpleBidEntry for the campaign's market segment that bids 1.0 and limits spending in the market segment to the campaign's budget:

```java
SimpleBidEntry bidEntry = new SimpleBidEntry(
    this.myCampaign.getMarketSegment(),
    1.0,
    this.myCampaign.getBudget());
```

Note: if your agent's market segment is more general, this entry will bid in all the specific market segments that are subsets of its market segment at the given price using the given limit. For example, if the campaign’s market segment is Female_Old, this bid entry will bid 1.0 with a spending limit of the campaign’s budget in both Female_Old_HighIncome and Female_Old_LowIncome.

Next, you would create a set of bid entries, and add this particular bidEntry to the set:

```java
Set<SimpleBidEntry> bidEntries = new HashSet<SimpleBidEntry>();
bidEntries.add(bidEntry);
```

Finally, you would create a OneDayBidBundle for your campaign that includes these bid entries and limits total spending to the budget.

```java
BidBundle bidBundle = new OneDayBidBundle(
    this.myCampaign.getId(),
    this.myCampaign.getBudget(),
    bidEntries)
```

**OneDayAgent Class**

Your job is to extend the abstract class OneDayAgent by implementing the getBidBundle() method. This method should return a OneDayBidBundle with all the agent’s bids for the game.

The class offers a Campaign object, myCampaign, which contains the campaign assigned to the agent at the beginning of the game.
Main Method

As in previous games, your agent needs a `main` method. For this game, the `main` method should create an agent object which in this case takes a nickname. Choose any nickname you like!

```java
public static void main(String[] args) {
    MyAgent myAgent = new MyAgent("localhost", 9898);
    myAgent.connect("CrazyNickName");
}
```

Helper Functions

Your agent might want to iterate over market segments. The `adxgame.jar` file contains support code for this. Concretely, each `MarketSegment` is implemented as an `Enum`, so to iterate through them you can do something like:

```java
for (MarketSegment m : MarketSegment.values()) { ... }
```

The static function `marketSegmentSubset(MarketSegment m1, MarketSegment m2)` returns a boolean indicating whether `m2` is a subset of `m1` (this is NOT a strict subset, so segments are subsets of themselves). You may wish to use this functionality as well, to construct a more sophisticated strategy.

Including the `adxgame.jar` File

The `adxgame.jar` file can be found in the course folder: `/course/cs1951k/pub/trading_platform/adxgame.jar` . As usual, to include it in your Eclipse project: Right-click -> Build Path -> Configure Build Path -> Libraries -> Add External JARs

Appendix

User Distributions

<table>
<thead>
<tr>
<th>Segment</th>
<th>Average Number of Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male_Young_LowIncome</td>
<td>1,836</td>
</tr>
</tbody>
</table>
### User Distributions: An Alternative View

<table>
<thead>
<tr>
<th></th>
<th>Young</th>
<th>Old</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>2,353</td>
<td>2,603</td>
<td>4,956</td>
</tr>
<tr>
<td>Female</td>
<td>2,236</td>
<td>2,808</td>
<td>5,044</td>
</tr>
<tr>
<td>Total</td>
<td>4,589</td>
<td>5,411</td>
<td>10,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Low Income</th>
<th>High Income</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>3,631</td>
<td>1,325</td>
<td>4,956</td>
</tr>
<tr>
<td>Female</td>
<td>4,381</td>
<td>663</td>
<td>5,044</td>
</tr>
<tr>
<td>Total</td>
<td>8,012</td>
<td>1,988</td>
<td>10,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Young</th>
<th>Old</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Income</td>
<td>3,816</td>
<td>4,196</td>
<td>8,012</td>
</tr>
<tr>
<td>High Income</td>
<td>773</td>
<td>1,215</td>
<td>1,988</td>
</tr>
<tr>
<td>Total</td>
<td>4,589</td>
<td>5,411</td>
<td>10,000</td>
</tr>
</tbody>
</table>

### Campaigns Distribution

A campaign targets one of the 20 possible market segments (a combination of at least two attributes) at random. The reach of the campaign is given by the average number of users in the selected segment (see table above) times a random discount reach factor selected from \(\{0.5, 0.6, 0.7\}\), where \(0 \leq 0.6 \leq 1\), for all i. The budget is always $1 per impression. The exact value of each discount factor depends on the number of people playing the game and thus, will be announced in lab.