Coffee App Notes

Algorithm Implementation

I started with implementing the basic version of the algorithm for choosing who should pay. The basic idea is to use the cost of the coffee as a weighting for choosing who pays. So, the more expensive relative to the other orders the more likely someone is to be chosen. In a simple example, everyone orders the same thing and has an equal weighting to be chosen. Whereas in an extreme case where 6 people chose the same thing, costing $1, and the 7th person chooses something costing $6, the 7th person is chosen around 50% of the time.

I did some manual tests of the results of this algorithm by calling it repeatedly and tabulating the results. In both cases I saw distributions that I would expect, i.e. in the even case everyone paid a similar amount of time (though not the same). And in the extreme case the person who got the most expensive item paid a majority of the time proportional to the relative cost of their item against the combined cost of everyone else.

Iterating on this algorithm, I wanted to protect against a person getting unlucky. In the base implementation one person could get the least expensive item and have to pay multiple days in a row due to the random nature of using a probability distribution. To protect against this I tracked how much each person has ordered and subtracted the amount this paid from that total amount whenever they paid. I used this rolling total as the weights for choosing who should pay, making sure to account for a negative balance (making it a zero weight). In this way, someone won't have to pay whenever they have paid in more than they have taken.

Doing the same manual tests on this enhanced version of the algorithm, I found that how often someone had to pay is more closely aligned to the proportion that their order costs against the entire order. In the even example, the algorithm would iterate through each person not going back to the same person until everyone else had paid. In the extreme example, the person who bought the drink costing 50% of the total paid exactly 50% of the time while everyone else paid an even amount spread between them.

The end version of the algorithm supports changing both the people and their orders day to day.

I decided to implement this code in python using the PayAlg.py class.

Backend Implementation

When considering how to service this algorithm to a UI, I decided to go with a simple backend server implemented in python. I looked at Django, Flask, and FastAPI since those were the project options in PyCharm. I decided to go with Flask since I had implemented a simple backend server using this library before and remembered it being fairly lightweight and easy to work with.

I initially created two endpoints. One that takes in a list of person, order pairs and returns who should pay. And another to see the current distribution of person, amount pairs.

I also modified the implementation to keep a running tally in memory so that it would keep a running total while it was running, but that information would get wiped between runs since it was being stored in memory instead of persistent storage. I considered some persistent storage solutions, but decided I would pursue that if I wanted to make some further enhancements if time permits. Two possible solutions I thought of was either a database solution (better for scalability) or writing a file to the disk (faster to implement but not scalable).

Frontend Implementation

I haven't done a lot of frontend work, but I wanted to try creating a webpage using React to get some hands-on experience. I added React to my project and tried connecting the frontend and backend first to make sure they could communicate. After I asked Gemini to create a webpage using react where I can input a person, order pairing into a list and have a button at the bottom to. Gemini then generated the code for the page that I ended up using. I made some tweaks to it and made sure to go over it so I understood what it was doing.