Coding Analysis
Nahid Alam

1. How long did you spend working on the problem? What did you find to be the most difficult part?
   3 hours for a working solution. Additional 2 hours for improving the heuristics. My current heuristics now produces higher expected yield than the sample output for the smaller dataset.

   The most difficult part was to find a good architecture of the program - what are the classes and objects, what other storage should I create beyond the class objects for quicker access etc.

2. How would you modify your data model or code to account for an eventual introduction of new, asofyet unknown types of covenants, beyond just maximum default likelihood and state restrictions?

   I will have a list of covenants for each facility object. While granting the loan, I can iterate over the list of covenants for that specific facility

3. How would you architect your solution as a production service wherein new facilities can be introduced at arbitrary points in time. Assume these facilities become available by the finance team emailing your team and describing the addition with a new set of CSVs

   Assuming my team converts the description of the finance team to more facilities.cvs file, the code will have the option to always wait for the next input files. More facility objects will be created per the new input files, maintaining the older ones.
4. Your solution most likely simulates the streaming process by directly calling a method in your code to process the loans inside of a for loop. What would a REST API look like for this same service? Stakeholders using the API will need, at a minimum, to be able to request a loan be assigned to a facility, and read the funding status of a loan, as well as query the capacities remaining in facilities.

   Request a loan be assigned -> POST request
   Read the funding status of a loan -> GET request
   Query the capability remaining -> GET request

   Sample NodeJS service interface

   ```
   app.get('/fundingStatus', function(req, res)){
   }

   app.get('/capabilityRemaining', function(req, res)){
   }

   app.post('/assignloan', function(req, res)){
   Loan_amount = req.body.loan_amount
   //do processing on the loan amount
   }
   ```

5. How might you improve your assignment algorithm if you were permitted to assign loans in batch rather than streaming? We are not looking for code here, but pseudo code or description of a revised algorithm appreciated.

   I will use 2-D 0/1 Knapsack algorithm in that case. Each column will represent each facilities and each row will work on optimizing the yield amount. The yield amount at any given cell will look into all previous yield amount and add its current expected yield if that leads to an optimal solution

   i. Given a facility with maximum loan capacity W, and a set S consisting of n loan ids.

   ii. Each loan id i has some loan amount w

   iii. For solving the problem we can generate the sequence of yield amount in order to obtain the optimum selection.

   iv. Let Xn be the optimum sequence of loan ids and there are two instances {Xn} and {Xn-1, Xn-2… X1}.

   Vi. So from {Xn-1, Xn-2… X1} we will choose the optimum yield amount with respect to Xn.

   vii. The remaining set should fulfill the condition of filling the facility of maximum loan amount W with maximum yield amount.

6. Because a number of facilities share the same interest rate, it’s possible that there are a number of equally appealing options by the algorithm we recommended you use (assigning to the cheapest facility). How might you tiebreak facilities with equal interest rates, with the goal being to maximize the likelihood that future loans streaming in will be assignable to some facility?

   My goal will be to assign as many loan as possible within the constraints. I will keep a system wide max of amount of the remaining loan a facility can provide. My goal will be to keep this system wide max as high as possible. To break the tie, I will chose the facility that allows me to keep the system wide max remaining loan giving capability as high as possible.

Notes on further improvements:

1. Modularizing the csv file read:
   a. I wanted to separate out the common file read portion for all the csv. But it showed error for loans.csv. I didn’t have time to debug further but would like to improve this. Below the common read logic for all the files:

```
def read(path):
   data_lines=[]
   f = open('large/facilities.csv')
   has_header = csv.Sniffer().has_header(f.read(1024))
   f.seek(0)  # Rewind
```

```
reader = csv.reader(f)
if has_header:
    next(reader)
for line in f:
    data_line = line.rstrip().split('\t')
    data_lines.append(data_line)
return data_lines
```

2. Modularizing the code even more:
    a. I did some heuristics improvement for maximizing the yield. I had to keep two global variables for that. Ideally I would like to encapsulate them in an object but didn't have time as I came up with the heuristics later. These are the two global variables

    ```
    max_amount_remaining = -1.0
    max_amount_remaining_facility_id = 0
    ```
3. Optimization – improving the heuristics:
    a. One approach of improving the heuristics is to keep a maximum amount of remaining loan amount for facilities per banned_state rather than keeping an overall system wide maximum amount of remaining loan amount. This will allow us to assign loans more optimally. This would require more structure changes so I didn't implement it.