Jerry Cain
CS106AX

Handout #27
December 8th, 2025

CS106AX Final Examination

General instructions

Answer each of the five questions included in the exam. Write all answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points is 100. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem. This leaves you with an additional 80 minutes to check your work or recover from false starts.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get a little bit of partial credit if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

First and Last Name: ____

SUNet ID (your `@stanford.edu` address): _______________________

Section Leader: ____

Problem 1: Python Strings (20 points)

Implement a function called `lar`—short for **l**ongest **a**lphabetic **r**un—that accepts a single string of just lowercase letters and returns the **longest substring** in which the characters appear in **increasing alphabetical order**. Restated, it should return the longest substring where each character is alphabetically larger than the one preceding it.

For instance, a call to `lar('barter')` would return `'art'`, since `'t'` comes after `'r'` comes after `'a'` in the alphabet, and there aren't any substrings of length 4 or more here that are sorted by character. A call to `lar('abhors')` would return `'abhors'`, since the incoming string is incidentally sorted from low character to high. If more than one qualifying substring has the same maximum length, `lar` can return any one of them.

More examples:

`lar('martyr')` *returns* `'arty'`
`lar('biopsy')` *returns* `'biopsy'`
`lar('spoonfeed')` *might return* `'s'`
`lar('')` *returns* `''`

Your implementation should **naively consider every possible substring** using a double `for` loop over its starting and ending indices. For each substring, determine whether it is in strictly increasing alphabetical order and update some best-so-far result when appropriate.

Present your implementation on the next page.

(space for your answer to problem #1 appears on the next page)

```
def lar(s):
   """
   Returns the longest substring of s whose characters are
   in strictly increasing alphabetical order. If more than
   one substring has the same maximum length, then any one
   of them can be returned.
   """
```

Problem 2: Python Lists (20 points)

You're to implement a function called **`csc`** (short for **`c`**onstruct **`s`**eating **`c`**hart) that, perhaps predictably, constructs a seating chart for a wedding reception knowing full well that many of the guests don't get along at all. This **`csc`** function takes three parameters:

- **`guests`**, a list of distinct guest names
- **`enemies`**, a list of two-element lists of the form [**`one`**, **`two`**], where **`one`** and **`two`** are the names of guests who should not be seated at the same table, and
- **`size`**, a positive integer that specifies the maximum number of guests per table

and returns a list of tables (each itself a list of guest names) such that:

- every guest appears exactly once in the return value
- all guests at any one table get along fine
- no table contains more than **`size`** guests
- each guest is seated in turn, from index 0 on up, using the following strategy:
  - try each of the existing tables and seat the guest at the first one that has space and contains none of the guest's enemies
  - if the guest can't be seated at any of the existing tables, start a new one with just that guest

For example, assuming that:

```
guests = ["Andy", "Tina", "Sabrina", "Jenny", "Diego", "Eugene"]
enemies = [["Andy", "Tina"], ["Andy", "Diego"], ["Tina", "Sabrina"],
           ["Sabrina", "Jenny"], ["Sabrina", "Eugene"], ["Diego", "Eugene"]]
size = 3
```

a call to **`csc(guests, enemies, size)`** might return

```
[['Andy', 'Sabrina'], ['Tina', 'Jenny', 'Diego'], ['Eugene']]
```

Present your implementation of **`csc`** on the next page. You may assume that **`size`** is always a positive integer, that all names in **`guests`** are distinct, and a name can only appear in **`enemies`** if it also appears in **`guests`**. The order of the tables in the outer list and the order of guests within each table does not matter.

(space for your answer to problem #2 appears on the next page)

```
def csc(guests, enemies, size):
   """
   Constructs and returns a seating arrangement according to the
   constraints specified as part of the problem statement.
   """
```

Problem 3: Working with Python Dictionaries (20 points)

Several music artists are temporarily removing their catalogs from Spotify due to contract and licensing changes. As a result, some songs will disappear from users' playlists on January 1. The company therefore needs a tool to analyze user playlists and determine how each of them will change as songs are pulled.

You're to implement **`prune`**, which accepts a user's **`playlists`** and a list of songs being **`removed`** from the platform. **`playlists`** itself is a dictionary, where the keys are playlist names and the values are lists of songs in that playlist. Your **`prune`** function should return a new dictionary mapping each playlist name to a **dictionary** with two entries:

- **`"removed"`**: the list of songs in the playlist being removed
- **`"retained"`**: the list of remaining songs in the playlist

So, assuming the following declarations:

```
playlists = {
   "Workout Mix": ["Titanium", "Levitating", "Bad Habits", "Halo"],
   "Road Trip": ["Africa", "Halo", "Believer"],
   "Chill": ["Easy On Me", "Someone Like You"]
}
removed = ["Halo", "Bad Habits"]
```

a call to **`prune(playlists, removed)`** would return

```
{
  "Workout Mix": {
      "removed": ["Bad Habits", "Halo"],
      "retained": ["Titanium", "Levitating"] },
  "Road Trip": {
      "removed": ["Halo"],
      "retained": ["Africa", "Believer"] },
  "Chill": {
      "removed": [],
      "retained": ["Easy On Me", "Someone Like You"] }
}
```

Your implementation shouldn't modify the incoming parameters, but it should preserve song order. If all songs in a playlist are retained, then **`removed`** should map to an empty list. Similarly, if all songs in a playlist are wiped out, then **`retained`** should map to an empty list.

Present your implementation of **`prune`** on the next page:

```
def prune(playlists, removed):
  """
  Accepts a dictionary mapping playlist names to
  lists of songs, along with a list of removed songs,
  and returns a new dictionary describing which songs
  were removed and retained in each playlist.
  """
```

Problem 4: Defining Python Classes and Reading Files (20 points)

You've recently joined OpenAI to work on the ChatGPT model integrity team. Every night, the team runs various ChatGPT models on a fixed collection of **evaluation prompts** and saves the model's token predictions to a text file for additional processing.

Each evaluation file is organized as a sequence of **blocks**, where each block corresponds to a single evaluation example and has this structure:

- the first line of the block is the example name (an arbitrary string), such as **`Example 101: subject line`**
- the lines that follow list the model's token predictions for that example
- a blank line, which marks the end of a block (including the last one)

Each prediction line has the form:

```
"token" probability model
```

For example, a small excerpt from the text file might look like this:

```
Example 101: subject line
" What a lovely idea!" 0.74 gpt-4.1
" Re: your idea"       0.18 gpt-4.1
" Following up"        0.08 gpt-4.1

Example 203: sentiment label
"positive" 0.83 gpt-4o
"neutral"  0.11 gpt-4o
"negative" 0.06 gpt-4o

Example 310: code completion
"\n    return result"          0.41 gpt-5.1
"\n    return None"            0.27 gpt-5.1
"\n    raise NotImplemented"   0.19 gpt-5.1
"\n    print(result)"          0.13 gpt-5.1
```

Here are some pertinent details:

- The **entire** first line of a block (e.g., Example 203: sentiment label) is the example's name.
- On each prediction line, the **`token`** is surrounded by double quotes. The token:
  - may contain spaces and punctuation but will never contain a double quote
  - may begin with a space (as in **`" What a lovely idea!"`**),
- **`probability`** is a floating-point number between 0.0 and 1.0.
- **`model`** is a single word naming the specific model that produced the prediction (e.g., **`gpt-5.1`**).

Assume the file exists and is properly formatted so you needn't do any error checking:

For this problem, you're to provide the implementation of a Python class called **`PredictionData`**, whose constructor opens a file, assumes it's structured as above, and parses the contents to build a suitable internal representation of the information that allows it to respond to two additional methods very, very quickly. Those methods are:

- **`getExampleNames`**, which returns a Python list containing all of the example names that appear in the file (for instance, **`"Example 101: subject line"`**, **`"Example 203:`**

sentiment label", and so on).

- **getPredictions**, which accepts an example name and returns all predictions for the provided name. The return value should be a list of dictionaries, where dictionary has keys **"token"**, **"probability"**, and **"model"**.

  So, a call to **getPredictions("Example 101: subject line")** would return:

  ```
  [
      { "token": " What a lovely idea!",
        "probability": 0.74,
        "model": "gpt-4.1" },
      { "token": " Re: your idea",
        "probability": 0.18,
        "model": "gpt-4.1" },
      { "token": " Following up",
        "probability": 0.08,
        "model": "gpt-4.1" }
  ]
  ```

  If the name doesn't appear in the file, **getPredictions** should just return the empty list to be clear there aren't any recorded predictions.

Using the next two pages, present your implementation of the **PredictionData** constructor and the two methods as described above. The constructor should do all the file processing and most of the work needed to ensure the implementations of the two methods are fairly short. You may use the **TokenScanner** if you'd like, or you can brute-force parse using string methods. Recall that you can convert a string called **str** to a **float** by calling **float(str)**.

(space for your answer to problem #4 appears on the next page)

```
class PredictionData:
    """
    Stores and provides access to LLM evaluation predictions.
    Each block in the input file corresponds to one evaluation example
    and contains a list of predicted next tokens with probabilities.
    """

    def __init__(self, filename):
        """
        Reads the evaluation file and stores all examples and predictions.
        The file is organized into blocks, where each block begins with an
        example name followed by one or more prediction lines.

        Each prediction line is converted into a dictionary with keys
        "token", "probability", and "model".
        """
```

(even more space for your answer to problem #4 appears on the next page)

```
def getExampleNames(self):
   """
   Returns a list of all example names that appeared in the file.
   """
```

```
def getPredictions(self, name):
   """
   Returns the list of predictions for the example name provided.
   If the example name doesn’t exist, the empty list is returned.
   """
```

Problem 5: Client-Side JavaScript (20 points)

You're writing an operating system component to poll all nearby Wi-Fi networks and connect to the first one that responds. To test whether a network is reachable, you send an HTTP **`GET`** request to:

```
GET <networkURL>/available
```

A successful request means the network is reachable. When the request succeeds, the server responds with a JSON object of the form:

```
{ "name": "Doris" }
```

Implement the **`connectToWiFi`** function, which accepts an array of network URLs called **`networks`** and immediately sends **`GET`** requests to **`<networkURL>/available`** for every URL in the **`networks`** array. As soon as the **first** request succeeds, your implementation should:

- connect to the Wi-Fi network by calling **`connect(networkURL)`**, where **`networkURL`** is the URL of the Wi-Fi network to respond first
- parse the JSON response payload
- print **`"Connected to Doris"`**, where **`"Doris"`** should be the name of the network in the JSON response payload (which won't always be Doris.)
- ignore all subsequent response by effectively doing nothing, which in practice means returning right away without connecting or printing anything

If none of the requests respond within 10 seconds, you should print **`"No networks available."`** and ignore all responses that eventually come in anyway. After all, your time is valuable and you shouldn't have to wait 10 seconds for Wi-Fi. ☺

Note that exactly one line of output should be printed, using **`console.log`**. Of course, you'll rely on the **`AsyncRequest`** and **`AsyncResponse`** classes introduced in lecture.

Using the space on the next page, present your JavaScript implementation of **`connectToWiFi`** that's coded to the requirements I just described. You'll also need to write a success handler, which should be an inner function that has access to **`connectToWiFi`**'s parameters and let variables. You don't need to write any error handlers.

(more space for your answer to problem #5 appears on the next page)

```
const DELAY = 10000; // max wait time in milliseconds
function connectToWiFi(networks) {
   // present your implementation in the space below
```