Jerry Cain
CS106AX

Handout #27S
December 10th, 2025

CS106AX Final Examination Solution

Everyone on course staff—Ben, Jenny, Sabrina, Eugene, Diego, Tina, and Andy—pressed through all of the final exam grading on Monday and Tuesday, and they are done. Only 29 of you absolutely *needed* to take the final exam, but ❤️ three contest winners took it anyway ❤️!

The final exercised everything we learned during the Python and server-side JavaScript material we've studied since Week 5, and as you all figured out, Problem 5 was tricky!! But in the face of tricky, you all did really, really well. The median grade was a 88 out of 100, and the mean was an 84. You can all head over to Gradescope at https://www.gradescope.com to see how you did.

To be fully transparent, here's how everyone did and the grade your final exam maps to.

Median = 88

Range	**Grade**	**N**
97–100	A+	3
90–97.5	A	13
86–89.5	A–	3
79–85.5	B+	4
62–78.5	B	9
50–61.5	B–	1

Expect to see final grades for the course posted no later than Tuesday evening, but probably much, much sooner than that.

Thanks for a great quarter and being such terrific students! It's fantastic that you're all so capable of completing such demanding assignments like Wordle, Enigma, Reassemble, RSG, and Adventure!, particularly given that so many of you are freshmen and may not have realized what you were signing up for!

Have a splendid winter break, and if you have the time, take CS106B as soon as you can!

Solution 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.

```
def is_increasing(t):
    """
    Returns True if t' characters are in strictly increasing
    alphabetical order (each char > previous one).
    """
    for i in range(1, len(t)):
        if t[i] <= t[i - 1]:
            return False
    return True

def lar(s):
    """
    Returns the longest substring of s whose characters are in strictly
    increasing order. If more than one substring has the same
    maximum length, then any one of them can be returned.
    """
    best = ""
    n = len(s)
    for start in range(n):
        for end in range(start + 1, n + 1):
            candidate = s[start:end]
            if is_increasing(candidate) and len(candidate) > len(best):
                best = candidate
    return best
```

Solution 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']]
```

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.

```
def can_sit_together(one, two, enemies):
    """
    Returns True if and only if one and two aren't enemies and
    can be seated at the same table
    """
    return [one, two] not in enemies and [two, one] not in enemies

def can_sit_at_table(guest, table, enemies, size):
    if len(table) >= size:
        return False
    for occupant in table:
        if not can_sit_together(guest, occupant, enemies):
            return False
    return True

def csc(guests, enemies, size):
    """
    Constructs and returns a seating arrangement according to the
    constraints specified as part of the problem statement.
    """
    tables = []
    for guest in guests:
        placed = False
        for table in tables:
            if can_sit_at_table(guest, table, enemies, size):
                table.append(guest)
                placed = True
                break
        if not placed:
            tables.append([guest])
    return tables
```

Solution 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.

```
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.
    """
    pruned = {}
    for name, songs in playlists.items():
        pruned[name] = {"removed": [], "retained": []}
        for song in songs:
            if song not in removed:
                pruned[name]["retained"].append(song)
            else:
                pruned[name]["removed"].append(song)
    return pruned
```

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.

Note: 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)**.

```
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".
    """
    self._names = []
    self._predictions = {}

    with open(filename) as infile:
        lines = infile.read().splitlines()
        count = 0
        while count < len(lines):
            name = lines[count]
            self._names.append(name)
            count += 1
            predictions = []
            while lines[count].strip() != "":
                start = lines[count].find('"') + 1
                stop = lines[count].find('"', start)
                token = lines[count][start:stop]
                rest = lines[count][stop + 1:].strip()
                spos = rest.find(' ')
                probability = float(rest[:spos])
                model = rest[spos + 1:].strip()
                predictions.append({
                    "token": token,
                    "probability": probability,
                    "model": model
                })
                count += 1
            count += 1
            self._predictions[name] = predictions

def getExampleNames(self):
    return self._names

def getPredictions(self, name):
    return self._predictions.get(name, [])
```

Solution 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.

```
const DELAY = 10000; // max wait time in milliseconds
function connectToWiFi(networks) {
    let found = false;
    let timedout = false;

    let onTimeout = function() {
        if (found) return; // optional if clearTimer is called
        timedout = true;
        console.log("No networks available.");
    }

    let timer = setTimeout(onTimeout, DELAY);
    for (let i = 0; i < networks; i++) {
        let network = networks[i];
        // need an per-iteration inner function to capture network URL.
        let onSuccess(response) = function {
            if (found || timedout) return;
            found = true;
            connect(network);
            let payload = JSON.parse(response.getPayload());
            console.log("Connected to " + payload.name);
            clearTimer(timer); // optional, not formally taught
        }

        let request = AsyncRequest("http:// " + network + "/available");
        request.setSuccessHandler(onSuccess);
        request.send()
    }
}
```