CS 106AX Final Exam Solution

I feel like everyone in CS106AX—the students, the section leaders, Doris—was a model student when it came to this final exam. The students did brilliantly, the section leaders graded the final exams with zero fuss, and Doris constructed this beautiful answer key. 😺

The final exam was out of 100 points—all problems were equally weighted—and the median grade was an 87.5. That was high enough that I didn’t really have to curve the exam as I did for the midterm, though I did curve up a bit so lower scores weren’t final grade killers.

I thought the final exam was challenging, so the fact that the median was so high was genuinely terrific. Congratulations on everyone for really showing up this quarter and doing so nicely across the board!

Final grades will have been posted by the time you’re reading this. 😊
Solution 1: Python Strings [20 points/100 total]
Write a Python function called \texttt{lbss}—short for longest \texttt{bookended} \texttt{substring}—that accepts a single string called \texttt{s} and returns the longest substring between two identical characters. If there are multiple such substrings, then you can return any one of them. And if there are no such substrings because there are no repeated characters in the supplied string, you should just return \texttt{None}.

\begin{itemize}
  \item \texttt{lbss("abcd\texttt{a}\texttt{d}")} would return \texttt{"bcd"}
  \item \texttt{lbss("ababacac")} returns \texttt{"babac"}
  \item \texttt{lbss("aaxabacacyadc")} returns \texttt{"axabacacy"}
  \item \texttt{lbss("aabbccddd")} returns \texttt{""}
  \item \texttt{lbss("abcdefgh")} returns \texttt{None}
\end{itemize}

Present your implementation in the space provided:

```python
def lbss(s):
    
    Analyzes the incoming string called \texttt{s} and
    returns the longest substring between any two
    identical characters.
    
    \texttt{def lbss(s):
    length = -1
    lss = None
    for left in range(len(s)):
        ch = s[left]
        right = s.rfind(ch)
        if right - left - 1 > length:
            length = right - left - 1
            lss = s[left + 1:right]
    return lss}
```
Solution 2: Python Lists [20 points/100 total]

Polydivisible numbers are positive integers such that the first k digits, when taken as a number, are divisible by k, for all reasonable values of k. For example, 8076 is a polydivisible number, because:

- 8 is divisible by 1
- 80 is divisible by 2
- 807 is divisible by 3
- 8076 is divisible by 4

Other examples of polydivisible numbers: 30080, 1652588, 444402009, 80480408404, and 76245056107220 are all examples of polydivisible numbers. The set of all polydivisible numbers is finite, and 3608528850368400786036725, at 25 digits, is the largest.

Here we describe an iterative algorithm that generates a sorted Python list of all polydivisible numbers. Our algorithm makes relies on the fact that, say, a nine-digit number can only be polydivisible if its first eight digits also comprise a polydivisible number. More generally, a k-digit number can only be polydivisible if its first k-1 digits comprise a polydivisible number as well.

Write a function called gpn (short for generatePolydivisibleNumbers) that programmatically generates all of the polydivisible numbers. Rather than bundle all numbers in a single list, return a list of 26 lists: the 0th list is the empty list (since there are no polydivisible numbers with zero digits), the 1st list is [1, 2, 3, 4, 5, 6, 7, 8, 9] (since all single digit numbers are divisible by 1), the 2nd list contains all two-digit numbers divisible by 2, the 3rd list contains all three digit numbers divisible by 3, and so forth, up to a 25th list that contains 3608528850368400786036725 (which is the only 25-digit number divisible by 25).

Your function should programmatically generate the entire list of 26 lists, where the list at index 1 is used to generate the list at index 2, which in turn is used to generate the list at index 3, and so forth. As you discover that, say, 807 is a polydivisible number, you’ll use that 807 to generate 8070, 8071, 8072, etc., though 8079—just multiply 807 by 10 and add 0 or 1 or 2, etc.—and include those that are incidentally polydivisible as well.

Present your implementation in the space provided:

```python
def gpn() :
    """
    Generates the full list of all polydivisible numbers, partitioned by digit count. gpn always returns
    the same thing, and that return value is:
    [
        [],
        [1, 2, 3, 4, 5, 6, 7, 8, 9],
```
numbers = []
numbers.append([])
numbers.append([1, 2, 3, 4, 5, 6, 7, 8, 9])
length = 2
while True:
    base = numbers[-1]
    extensions = []
    for num in base:
        for d in range(10):
            candidate = 10 * num + d
            if candidate % length == 0:
                extensions.append(candidate)
    if extensions == []: return numbers # yes? it's over
    numbers.append(extensions)
    length += 1
Solution 3: Working with Python Dictionaries [20 points/100 total]

Credit card fraud is a rampant problem plaguing the banking industry, and many companies in recent years—Stripe, Venmo, Affirm, and many others—have gained traction as payment platforms because of added security measures they claim to implement.

Credit cards, of course, come with credit card numbers, and the numbers themselves reveal information about the issuing bank—specifically, the first one to six numbers uniquely identify the bank that issued the card. The following Python dictionary models the direct association between prefixes and the names of the institution to which that prefix has been assigned (by the American Banking Association for the prefixes used in this problem):

```
CREDIT_CARD_PREFIXES = {
    '1': 'UATP',
    '31': 'China T-Union',
    '34': 'American Express',
    '37': 'American Express',
    '4': 'Visa',
    '51': 'Mastercard',
    # many more
    '8600': 'UzCard',
    '9860': 'Humo'
}
```

Each of the named banking institutions—American Express, China T-Union, Humo, etc.—maintain their own databases of valid credit card numbers and the credentials that must be presented if an attempt to use the credit for a purchase is to be approved. For simplicity, we’ll assume all such credit card numbers are accessible on a per-institution basis via another Python dictionary that looks like this:

```
CREDIT_CARD_NUMBERS = {
    'American Express': {
        'length': 15,
        'accounts': {
            '347218283219443': {'expires': '2020-03', 'zipcode': '13158', 'cvv': '492'},
            '379764432192032': {'expires': '2024-03', 'zipcode': '49823', 'cvv': '880'},
            # additional credit card records
        },
    },
    'China T-Union': {
        'length': 16,
        'accounts': {
            '3129139219339221': {'expires': '2006-05', 'zipcode': '08077', 'cvv': '122'},
            '3139493912392123': {'expires': '2026-10', 'zipcode': '19152', 'cvv': '194'},
            # additional credit card records
        },
    },
    # additional entries for all other institutions
}
```
Given the above data structures, you’re to write two functions—the first of which will be used to implement the second—to access relevant information that can be used to approve or decline a transaction. While writing these functions, assume that:

- all numbers are expressed as strings, since leading zeros matter
- all month-and-date strings are formatted as "YYYY-MM" so they can be compared using ==, <, etc.
- the keys of the accounts dictionaries are always full credit card numbers

a. [8 points] Write the `lookupBank` function, which examines the supplied credit card number and returns the name of the banking institution capable of issuing that number. You needn’t check whether the number is of the correct length or in the second dictionary. You only need to return the bank name and that’s it (or None if the credit card fails to match any of the prefixes). Your implementation can directly reference the `CREDIT_CARD_PREFIXES` global defined above.

```python
def lookupBank(number):
    
    """
    Returns the name of the bank owning the supplied credit card number (which is expressed as a string). If the credit card number doesn’t begin with one of the supported prefixes, lookupBank should return None.
    """
    for key in CREDIT_CARD_PREFIXES:
        if number.startswith(key): return CREDIT_CARD_PREFIXES[key]
    return None
```

b. [12 points] Write the `isApproved` function that, given a credit card number, the current month and year (expressed as "YYYY-MM" like all others in this problem), the supplied expiration date, cvv (card verification value), and zip code, returns True if and only if the credit card number exists, the card hasn’t expired, and all of the supplied credentials match those stored by the bank (and False otherwise). Your implementation can access either or both of `CREDIT_CARD_PREFIXES` and `CREDIT_CARD_NUMBERS`. Note that not all properties-keys need be referenced by your implementation, but you do need to guard against the possibility that the credit card number is malformed or isn’t one issued by any of the banks in `CREDIT_CARD_PREFIXES`. 
def isApproved(number, now, supplied_expires, supplied_cvv, supplied_zipcode):
    
    Returns True if and only if the number is valid, the current month and year (shared via now) is such that the card hasn’t expired, and the supplied expiration date, cvv code, and zip code all match the bank’s records.

    bank = lookupBank(number)
    if bank is None: return False

    accounts = CREDIT_CARD_NUMBERS[bank]['accounts']
    if number not in accounts: return False
    info = accounts[number]
    return now <= info['expires'] and supplied_expires == info['expires'] and 
    supplied_cvv == info['cvv'] and supplied_zipcode == info['zipcode']
Solution 4: Defining Python Classes and Reading Files [20 points/100 total]

As social media has flourished over the past two decades, so has the ability to spread misinformation and disinformation, collectively referred to as Fake News. Web properties like Facebook, Instagram, Twitter, and Flutterer build frameworks to identify a large fraction of misleading news articles and what isn’t caught by those frameworks is often flagged by the users.

There are a large number of datasets cataloguing news articles as either fake or real, and one such dataset—compiled from a snapshot of all tweets on Twitter with attached link during a window of a few weeks in 2018—is presented below:

Gossip Girl 10 Years Later: How Upper East Siders Shocked the World
https://www.zerchoo.com/entertainment/gossip-girl-10-years-later
www.zerchoo.com 38 1

Gwen Stefani Got Dumped by Blake Shelton Over "Jealousy and Drama"
https://www.intouchweekly.com/posts/gwen-stefani-dumped-156076
www.intouchweekly.com 45 0

Broward County Sheriff Fired For Lying
https://yournewswire.com/broward-county-sheriff-fired/
yournewswire.com 124 0

*additional groups all four lines, one group per tweet*

Information about each tweet is stored in a three-line entry, with a single blank line following it. Even the last entry has a single blank line afterwards.

The first line of each entry is the title of the tweeted article, and the second line is the article URL. The third line is a three, space-delimited string of three data items: the domain name hosting the article, the number of retweets, and a Boolean value expressed as an integer (0 for fake, 1 for real).

Implement a Python class called `LinkDatabase` whose constructor looks like so:

```python
class LinkDatabase:
    def __init__(self, filename):
```

The constructor should read the contents of the named file and compile its contents into a suitable internal form. Beyond the constructor, your class implementation should also support:

- A method called `getArticleInfo`, which accepts a URL and returns a small dictionary containing the article’s title, the retweet count, and whether the article is factual or fake, expressed as a `True` or `False`. Your dictionary should go with property/key names of `title`, `count`, and `verified`. If the URL doesn’t exist, then simply return `None`.
- A method called `getSiteReliability`, which returns the fraction (e.g., 0.94) of articles hosted by the supplied domain name that are deemed factual (or `None` if the domain name doesn’t appear anywhere).
For this problem, all you have to do is read the data into the internal structure. Any actual analyses are the responsibility of those using your LinkDatabase class. Present the full implementation of the class, which includes the constructor and the two methods described above, in the space provided. The vast majority of your code should reside within the constructor, and the implementations of the two additional methods should be very, very short and do little more than access pre-compiled information.

```python
class LinkDatabase:
    def __init__(self, filename):
        with open(filename, "r") as f:
            self._articles = {}
            self._reliabilities = {}
            lines = f.read().splitlines()
            for i in range(0, len(lines), 4):
                title = lines[i]
                url = lines[i + 1]
                domain, count, verified = lines[i + 2].split()
                self._articles[url] = {
                    'title': title,
                    'count': int(count),
                    'verified': verified == '1'
                }
                if domain not in self._reliabilities:
                    self._reliabilities[domain] = [0, 0]
                self._reliabilities[domain][int(verified)] += 1

    def getArticleInfo(self, url):
        if url not in self._articles: return None
        return self._articles[url]

    def getSiteReliability(self, domain):
        if domain not in self._reliabilities: return None
        num = self._reliabilities[domain][1]
        denom = self._reliabilities[domain][0] + self._reliabilities[domain][1]
        return num / denom
```
Solution 5: Client-Side JavaScript [20 points/100 total]

You’ve recently launched a commercial version of your Flutterer assignment, and reception has been overwhelmingly positive. Your users, however, are craving video support! To that end, you’ve added two new API endpoints to your Flutterer server:

- **POST api/upload**, which accepts as payload the contents of a video file and, on success, responds with a small JSON object containing nothing more than an id number forever associated with the video, and with:
  
  ```json
  { "id": 26172831 }
  ```

  The server responds as quickly as possible with the response, even though it may take several minutes to process the video and efficiently store it.

- **GET api/upload/<id>/status** assume the embedded id number is a valid video upload id and responds with its own JSON object containing the relevant video id and a percent field, as with:

  ```json
  { "id": 26172831, "percent": 14 }
  ```

  The above response suggests that the video with the stated id number is 14% processed, and that you should issue the same query again to get an updated progress percentage. You can assume that the percent value is always some integer between 0 and 100 inclusive.

Note that neither of the two endpoints require any query parameters (i.e., you never need to call `setParam` or `setParams`).

For this problem, you’d simply like to test both endpoints to see that they work properly. To do so, you’re to implement the `testVideoUpload` function, which accepts the video data as a string and issues the **POST** request with the supplied video string as payload. Your success handler should print "Video (id: 26172831) upload initiated." to the console before calling `setTimeout` to prompt a **GET** request (to the second of the two new endpoints using the id number it just received) be sent five seconds later. The success handler for the **GET** request should `console.log` the video id and the percentage on a single line—structure the line as "26172831: 14% processed."—and schedule another **GET** request for the same id be sent five seconds later if the percentage is anything less than 100%. Each success handler will schedule the same **GET** request to be sent five seconds into the future until the server responds saying the video has been fully processed at 100%.

Assuming video is a variable bound to the video data—yes, as one very long string of data, a call to `testVideoUpload(video)` via the execution of many success handlers five seconds apart, might print this over the course of approximately 30 seconds:

```
Video (id: 26172831) upload initiated.
26172831: 14% processed.
```
Of course, the id number will vary and will be dictated by the POST response. And the percentages would vary as well.

Use the space provided to implement your `testVideoUpload` function. Note that there’s no client-side DOM manipulation in this problem. We’re concerned primarily with your ability to use the `AsyncRequest` and `AsyncResponse` JavaScript classes discussed in lecture. Note that you’ll need to implement several small functions to serve as success handlers and timer functions. You need not, however, worry about any error checking anywhere.

```javascript
function testVideoUpload(video) {
  let onSuccessStatus = function(response) {
    let info = JSON.parse(response.getPayload());
    console.log(info.id + " : " + info.percent + "% processed.");
    if (info.percent === 100) { return; }
    setTimeout(function() {
      monitorUpload(info.id);
    }, 5000);
  };

  let monitorUpload = function(id) {
    AsyncRequest("api/upload/" + id + "/status")
      .setSuccessHandler(onSuccessStatus)
      .send();
  };

  let onSuccessUpload = function(response) {
    let info = JSON.parse(response.getPayload());
    console.log("Video (id: " + info.id + ") upload initiated.");
    setTimeout(function() {
      monitorUpload(info.id);
    }, 5000);
  };

  AsyncRequest("api/upload")
    .setMethod("POST")
    .setPayload(video)
    .setSuccessHandler(onSuccessUpload)
    .send();
}
```