CS106AJ Final Examination

Name (please print) ________________________________

Section Leader ________________________________

General instructions

Answer each of the questions given below. Write all of your 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 that the number of points be roughly equivalent to the number of minutes someone who is completely on top of the material would spend on that problem. Even so, we realize that some of you will still feel time pressure. If you find yourself spending a lot more time on a question than its point value suggests, you might move on to another question to make sure that you don’t run out of time before you’ve had a chance to work on all of them.

In all questions, you may include functions or definitions that have been developed in the course by giving the name of the function and the handout or chapter number in which that definition appears.

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.

THE STANFORD UNIVERSITY HONOR CODE

A. The Honor Code is an undertaking of the students, individually and collectively:
   (1) that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
   (2) that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

B. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid as far as practicable, academic procedures that create temptations to violate the Honor Code.

C. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

I acknowledge and accept the Honor Code.

(signed) ________________________________
Problem 1—Short answer (10 points)
1a) Suppose the perplexity function is defined as follows:

```javascript
function perplexity(array) {
    let befuddled = array[0];
    let baffled = [befuddled];
    for (let i = 1; i < array.length; i++) {
        befuddled = Math.max(array[i], befuddled + array[i]);
        baffled.push(Math.max(befuddled, baffled[i - 1]));
    }
    return baffled;
}
```

A quick glance of the code suggests the above function considers every element of the incoming array and returns a second array whose length is equal to that of the incoming one. Save for the 0th element, the for loop visits every element in array and ensures that some new element is introduced to the one that’s ultimately returned.

```
perplexity([-2, 1, -3, 4, -1, 2, 1, -5, 7, -10]);
```

Consider the execution of the above call to perplexity, and complete the diagram below to indicate the integer values present in the array return value.

```
[ ] [ ] [ ] [ ] [ ] [ ]
```

1b) What value is printed if you call the function TestPuzzle in the following code:

```javascript
function TestPuzzle() {
    console.log(puzzle(3, 4).enigma(5, 6));
}

function puzzle(x, y) {
    return {
        x: x * y,
        enigma: function(w, z) {
            return this.x + 100 * x + "0" + w + z;
        }
    };
}
```

Answer to Problem 1b: ________________
Problem 2—Simple graphics (15 points)

*Likert-scale surveys* generally offer those taking the survey are presented with exactly five ways to respond to a question. Two of the five are negative (e.g. "strongly disagree" and "disagree", or "much worse" and "somewhat worse"), one is neutral (e.g. "neither agree nor disagree", or "about the same"), and two are positive ("agree" and "strongly agree", or "better" or "much better").

One of the best ways of displaying Likert-type data—in particular, the data gathered from Likert-scale surveys—is the *diverging stacked bar chart*, or *Likert chart*. The following image, lifted from [http://rnotr.com/likert/ggplot/barometer/likert-plots/](http://rnotr.com/likert/ggplot/barometer/likert-plots/), uses Likert chart to summarize responses to single question drawn from the 2012-2014 Arab Democracy Survey.

![Likert Chart Image](http://rnotr.com/likert/ggplot/barometer/likert-plots/)

Note that each of twelve countries is paired with a single Likert chart, and all of the charts—or at least the parts that are visibly colored—are equally wide. However, those summarizing more negativity than positivity are shifted to the left, and those summarizing more positiveness than negativity are shifted to the right. Restated, the twelve charts are shifted so that the centers of each neutral segments in each stacked bar chart are aligned horizontally. The visual cues that come from the shifts left or right immediately convey that Lebanon is less optimistic about the future than Kuwait is.

For this problem, you’re to implement a function called `createLikertChart`, which accepts an array of length 5 such that the first element stores the percentage of participants who responded most negatively, the second element stores the percentage of participants who responses somewhat negatively, and so forth. Your `createLikertChart` should return a 200px by 20px, gray-bordered `GCompound`, and housed within the `GCompound` are the five rectangles—with colors "Red", "Orange", "LightGray", "Green", and "Blue"—drawn side by side such that the center of the
entire \texttt{GCompound} overlays the center of the gray rectangle. Note that the total width of the five stacked rectangles is always 100px, so that precisely half the area within the surrounding rectangle is occupied.

For example, a call to \texttt{createLikertChart([10, 10, 10, 43, 27])} would return a \texttt{GCompound} that looks like this:

\begin{center}
\includegraphics[width=0.3\textwidth]{example1.png}
\end{center}

Calls to \texttt{createLikertChart([0, 0, 0, 30, 70])}, \texttt{createLikertChart([25, 15, 10, 20, 30])}, and \texttt{createLikertChart([90, 0, 0, 5, 5])} would each return \texttt{GCompound}s that look like those below:

\begin{center}
\includegraphics[width=0.3\textwidth]{example2.png}
\end{center}

\begin{center}
\includegraphics[width=0.3\textwidth]{example3.png}
\end{center}

You might argue that two of the last three Likert charts don’t have gray rectangles. But I’ll argue that they do: They’re just of width zero, and their invisible centers still overlay the center of the entire Likert chart.

Use the next page to present your implementation of \texttt{createLikertChart}, relying on the constant definitions provided.

\textbf{(space for the answer to Problem 2 appears on the next page)}
Answer to Problem 2:

```javascript
const LIKERT_CHART_WIDTH = 200;
const LIKERT_CHART_HEIGHT = 20;
const LIKERT_CHART_BORDER = "Gray";
const LIKERT_CHART_COLORS = ["Red", "Orange", "LightGray", "Green", "Blue"];

function createLikertChart(percentages) {
```
Problem 3—Interactive graphics (20 points)
For this problem, you’re to implement an easily described, mouse-event-influenced animation of where a figurative cat chases a literal mouse.

Initially, a small, silver circle—that’s the figurative cat—is added to the graphics window so that it’s perfectly centered. The cat sits perfectly still until the user clicks the mouse, at which point the cat moves a distance of exactly one pixel—every 4 milliseconds—toward the mouse. As the mouse moves about the graphics window, the cat changes course accordingly and steadily moves 1px—again, every 4 milliseconds—in a direction that most efficiently closes the gap between the cat and mouse. This cat and mouse game continues unabated until the cat catches the mouse (or, restated, the mouse location resides somewhere within the circle), at which point the cat dies and goes to cat heaven, the animation ends, and nothing else of interest happens.

You’re to implement this animation, subject to the following constraints:

- Aside from the placement of a small, centered circle when the program begins, absolutely nothing else happens until the user clicks somewhere in the screen.
- The first mouse click initiates the only timer function to ever be installed, and that time function should figure out how to move the circle a distance of exactly one pixel in the direction where the mouse was last detected. Restated, at the mouse moves around the screen to avoid capture, you must maintain information about the mouse’s whereabouts so the code running the animation can respond to the mouse’s ever-changing location.
- Any additional mouse clicks should be ignored. In particular, you should be careful to not spawn a second animation that competes with the first.
- When the cat catches the mouse, the cat should be removed from the window, the animation should be terminated, and the program should effectively end.
- Use the Pythagorean theorem to compute distance. Recall that if $dx$ represents the difference between the cat’s and mouse’s x coordinates, and $dy$ represents the difference between the cat’s and mouse’s y coordinates, then the square of the Euclidean distance between the cat and mouse is the sum of the square of $dx$ and the square of $dy$.

Use the next page to present your implementation of the entire `CatAndMouse` program, relying on the constant definitions provided.

(space for the answer to Problem 3 appears on the next page)
Answer to Problem 3:

/* Constants */
const GWINDOW_WIDTH = 700;
const GWINDOW_HEIGHT = 400;
const CAT_RADIUS = 10;
const CAT_COLOR = "Silver";
const DELAY = 4;

function CatAndMouse() {
Problem 4—Strings (15 points)

Consider the naïve encryption scheme, where a lowercase word in the English language is transformed into an array of integers, where each letter of the encrypted word is represented by its position in the lowercase English alphabet. So, according to this encryption scheme, the word "oxygenate" would be encrypted—if you can call it that—as [14, 23, 24, 6, 4, 13, 0, 19, 4]. That’s because "o" is the 14th letter of the English alphabet (if we call "a" the 0th instead of the 1st letter), "x" is the 23rd letter, "y" is the 24th letter, and so forth. Given just the integer array, we could probably infer the encryption scheme from just one example, and reverse engineer a decryption scheme to easily recover "oxygenate" from [14, 23, 24, 6, 4, 13, 0, 19, 4], "onomatopoeia" from [14, 13, 14, 12, 0, 19, 14, 15, 14, 4, 8, 0], and "uncopyrightable" from [20, 13, 2, 14, 15, 24, 17, 8, 6, 7, 19, 0, 1, 11, 4].

Consider a slightly more elaborate encryption where words are converted to integer arrays as before, except that the integers are offsets into an ever-evolving substitution-cipher-like key, which is initially set to be the traditional alphabet of lowercase letters. As each letter of the incoming is converted to its integer offset, the letter is spliced out of the key and prepended to the front. The update key contains the same 26 lowercase letters, of course, except all letters those preceding the most recently encrypted one have shifted up one position to close the gap left by the letter hoisted to the front. This new key is used for the next letter just as the old one was for the prior letter, and is repeated updated with each character transformation. (This process has a side effect of bringing recently surfaced letters to the front of the key and pushing long-forgotten ones to the back.)

Given the new encryption scheme, the word "ooze" would be transformed into [14, 0, 25, 6], and here’s why:

- The key is initially set to the lowercase alphabet: "abcdefghijklmnopqrstuvwxyz".
- The first "o" is at index 14 of the key, and the key is transformed from "abcdefghijklmnopqrstuvwxyz" to "oabcdefghijklmnopqrstuvwxyz". The "o" was lifted out and prepended to the front.
- The second "o" is at index 0, since it was just hoisted to the front of the key. The key remains the same.
- "z" is at index 25 of the key. The "z" is then lifted out and moved to the front to produce yet another key, "zoabcdefghijklmnopqrstuvwxyz".
- Finally, "e" is at index 6 of the current key, and even though it won't matter—after all, this is the last letter—the key is updated to be "ezoabcdefghijklmnopqrstuvwxyz".

Recovery begins with the lowercase alphabet as the key, and the key is updated in precisely the same way. The primary difference is that offsets are converted to letters, whereas encryption converts letters to offset. To see how "ooze" is recovered from [14, 0, 25, 6], read through the following:

- The key is initially set to the lowercase alphabet: "abcdefghijklmnopqrstuvwxyz".
• The first offset is 14, and the 14\textsuperscript{th} character of the key is "o". As with the original encryption, the "o" is promoted to the front of the key to produce an updated one of "oabcdefghijklmnopqrstuvwxyz".
• The second offset is 0, and that’s the leading character of the current key: "o". We should know that, because we just put the "o" there one bullet point prior.
• The third offset is 25, and the key's 25\textsuperscript{th} letter is "z". The "z" takes the lead and bubbles to the front of the key, which is now "zoabcdefghijklmnopqrstuvwxyz".
• The final offset is 6, and as luck would have it, the key’s 6\textsuperscript{th} position is occupied by "e". The key is transformed so that "e" is at the front, but it doesn’t matter what they key is at this point, because the original "ooze" has been fully recovered.

Converting strings to offset arrays requires array operations like push, but recovering strings from offset encoding according to this new scheme doesn’t require any array operations other than selection.

You’re to use the next page to implement the \texttt{recover} function, which accepts an array of offsets into an ever-evolving substitution-cipher-like key and returns the word encoded by those offsets.

\textbf{(space for the answer to Problem 4 appears on the next page)}
Answer to Problem 4:

```javascript
const ALPHABET = "abcdefghijklmnopqrstuvwxyz";
function recover(encoding) {
```
Problem 5—Arrays (10 points)
Implement a function called dedupe, that accepts an arbitrary array and updates it so that all duplicates have been removed. dedupe doesn't return a new array, but instead updates the supplied one so that all duplicates are gone. As each duplicate element of the incoming array is removed, the length of the array shrinks by one. (The items that do remain need not appear in any particular order.)

Use the rest of this page to supply your implementation of dedupe.

(space for the answer to Problem 5 appears below)

function dedupe(array) {

Problem 6—Working with data structures (15 points)

Ribonucleic acid—more commonly known as RNA—is a biological polymer of nucleotides, where each nucleotide consists of a five-carbon sugar (specifically, a ribose), one or more phosphate groups, and a nitrogen-containing molecule that has the chemical properties of a base.

Each nitrogenous base can be one of four different molecules: uracil, adenine, guanine, and cytosine, generally abbreviated as U, A, G, and C, respectively. While RNA is really a polymer of alternating sugars and phosphate groups, where these nitrogenous bases extend off of the polymer, RNA is best viewed—at least for the purposes of this problem—as a sequence of U’s, A’s, G’s, and C’s, because that sequence dictates how RNA operates on behalf of the larger organism around it.

Proteins, like RNA strands, are also polymers, but they are polymers of molecules called amino acids. A protein’s three-dimensional structure and its specific biological function—maybe it’s an antibody, maybe it’s an enzyme, maybe it’s something else—are determined by its amino acid sequence. And while there are only four different nucleotides in RNA strands, there are 20 different amino acids, with names like alanine, cysteine, glutamic acid, and tryptophan.

Substrings of RNA called genes effectively instruct living cells how to synthesize proteins. Here’s a simplified version of how RNA encodes protein structure:

• Each gene is taken to be a sequence of nucleotide triplets called codons, and each codon is taken as an instruction to include some amino acid in a larger protein. If two codons are side by side in a gene, then the amino acids they encode are side by side in the corresponding protein. So, a gene fragment like "AGAUGGUGC" is really a sequence of three codons, "AGA", "UGG", and "UGC". "AGA" maps to arginine, "UGG" maps to tryptophan, and "UGC" maps to cysteine. If a gene’s first nine nucleotides—restated, three codons—are "AGAUGGUGC", then the encoded protein would begin with arginine, tryptophan, and cysteine.

• Because there are $4^3 = 64$ different codons and a mere 20 amino acids, many of the codons are treated synonyms and encode the same amino acid.

• The beginning of every gene is marked by the start codon, which is always "AUG". As it turns out, the amino acid methionine is also encoded by "AUG", but that mapping is only active after we’ve seen "AUG" the first time.

• The end of every gene is marked by any one of three different stop codons: "UAA", "UGA", "UAG".

The JavaScript map constant below contains 20 keys, each of which is the name of some amino acid. Each key is mapped to an array of all those codons that encode it.
const START_CODON = "AUG";
const STOP_CODONS = ["UAA", "UGA", "UAG"];
const MAPPINGS = {
  "alanine": ["GCU", "GCC", "GCA", "GCG"],
  "arginine": ["CGT", "CGC", "CGA", "CGG", "AGA", "AGG"],
  "asparagine": ["AAU", "AAC"],
  "aspartic acid": ["GAU", "GAC"],
  "cysteine": ["UGU", "UGC"],
  "glutamine": ["CAA", "CAG"],
  "glutamic acid": ["GAA", "GAG"],
  "glycine": ["GGU", "GGC", "GGA", "GGG"],
  "histidine": ["CAU", "CAC"],
  "isoleucine": ["AUU", "AUC", "AUA"],
  "leucine": ["UUU", "UUC", "CUU", "CUC", "UUA", "CUA", "CUG"],
  "lysine": ["AAA", "AAG"],
  "methionine": ["AUG"],
  "phenylalanine": ["UUU", "UUC"],
  "proline": ["CCU", "CCC", "CCA", "CCG"],
  "serine": ["UCU", "UCC", "UCA", "UCG", "AGU", "AGC"],
  "threonine": ["ACU", "ACC", "ACA", "ACG"],
  "tryptophan": ["UGG"],
  "tyrosine": ["UAU", "UAC"],
  "valine": ["GUU", "GUC", "GUA", "GUG"],
};

Using the next page, write a function called `mappingIsValid`, which accepts a gene and an array of amino acids, and returns `true` if and only if the provided gene begins with the start codon "AUG", ends with one of the three stop codons "UAA", "UGA", or "UAG", and everything in between precisely encodes the supplied sequence of amino acids and nothing else.

(space for the answer to Problem 6 appears on the next page)
Answer to Problem 6:

```javascript
function mappingIsValid(gene, sequence) {
```
Problem 7—Reading data structures from embedded XML (15 points)

Tag clouds are data visualizations often used to convey information about the most prominent words in a large data set (e.g. presidential speeches, college admissions essays, New York Times news articles, or leaked diplomatic cables). The most important, common, and impactful words are typically drawn in larger fonts sizes, and less compelling words, while important enough to be included, are drawn less prominently in smaller font sizes.

Assume that a large subset of all U.S. Presidential speeches has been parsed, and word tag data for all of those speeches has been embedded as XML within the surrounding index.html document, and that the XML has been formatted as follows:

```xml
<cloud id="CloudData" style="display:none;">  
  <speech title="Foundation of Government" date="1776-01-15">  
    <tag word="affections" color="#A19A7E" weight="15"/>
    <tag word="agreeable" color="#BBB6A2" weight="15"/>
    <tag word="ambition" color="#D9D6CB" weight="15"/>
    <tag word="ammunition" color="#CFCCBE" weight="19"/>
    <tag word="antiquity" color="#8C8361" weight="15"/>
    <tag word="approve" color="#D4D0C4" weight="15"/>
    <tag word="arcadia" color="#7B714A" weight="15"/>
    <tag word="aristocratical" color="#A8A188" weight="15"/>
    <tag word="arms" color="#DAD7CC" weight="15"/>
    <tag word="assembly" color="#DFDCD3" weight="47"/>
    // more tag elements
  </speech>
  <speech title="Importance of Property for the Suffrage" date="1776-05-26">  
    <tag word="accommodate" color="#CECABC" weight="19"/>
    // more tag elements
  </speech>
  <speech title="Debate on Independence" date="1776-06-07">  
    <tag word="absolved" color="#ADA78F" weight="13"/>
    // more tag elements
  </speech>
  // more speech elements
</cloud>
```

The entire dataset is anchored by a single `<cloud>` element with an id of "CloudData". All of the `<cloud>` element’s direct children are `<speech>` elements, where each `<speech>` element contains two attributes: the title of the speech, and the date the speech was delivered. Each `<speech>` element has one or more children, all of which are `<tag>` elements. Each `<tag>` element contains three attributes: the word of prominence, the color that should be used to render the word, and the font size that should be used to render the word. The accumulation of all `<tag>` elements under a single `<speech>` tag are alphabetically ordered by their `<word>` element.
For this problem, you’re to provide the full implementation of the **PresidentialWordCloud** factory function, which makes a single pass over the XML to build an internal representation the relevant data and returns an object with precisely two methods as fields, and nothing else. The two methods are:

- **getAllWords**, which accepts **title** and **date** strings as parameters, and returns the alphabetically sorted list of words that would contribute to that speech’s word cloud, and

- **getAllTags**, which accepts **title, date, and size** parameters, and returns an array of JavaScript aggregates, alphabetically sorted by word, for all tagged words in the speech identified by the supplied title and date that should be rendered in the supplied font size. Each of the objects in the returned array should be structured to contain precisely two fields: **word** and **color**. For example, a call to **getAllTags(“Foundation of Government”, ”1776-01-15”, 15)** would produce the following array as a return value:

```javascript
[
  {"word": "affections", "color": "#A19A7E"},
  {"word": "agreeable", "color": "#BBB6A2"},
  {"word": "ambition", "color": "#D9D6CB"},
  {"word": "antiquity", "color": "#8C8361"},
  {"word": "affections", "color": "#A19A7E"},
  // several similarly structured objects omitted for brevity
  {"word": "youth", "color": "#DEDCD2"}
]
```

Use the next two pages to present your implementation of the **PresidentialWordCloud** function. Be sure to make only one pass over the XML so that the executions of the two exported methods can quickly return without any additional XML data processing.

*(space for the answer to Problem 5 appears below)*
Answer to Problem 7:

function PresidentialWordCloud() {

Answer to Problem 7 (continued)