Jerry Cain — December 10, 2018
CS 106AJ — Time limit: 3 hours

CS106AJ Final Examination

Name (*please print*) ________________________________

Section Leader ________________________________

	score	initials
1	____/10	____
2	____/15	____
3	____/20	____
4	____/15	____
5	____/10	____
6	____/15	____
7	____/15	____
Total	____/100	

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 **conundrum** function is defined as follows:

```
function conundrum(str) {
   let len = 0;
   let result = [0];
   for (let i = 1; i < str.length; i++) {
      if (str.charAt(i) === str.charAt(len)) {
         len++;
         result.push(len);
      } else if (len === 0) {
         result.push(0);
      } else {
         len = result[len - 1];
         i--;
      }
   }
   return result;
}
```

A quick glance of the code suggests the above function crawls over the supplied string and returns an array of numbers. Except for index 0, the **for** loop examines every character in **str** and builds up an integer array whose length ultimately matches that of **str** itself.

Consider the execution of a call to **conundrum("AAACAAAAAC")** and complete the diagram below to indicate the integer values present in the returned result.

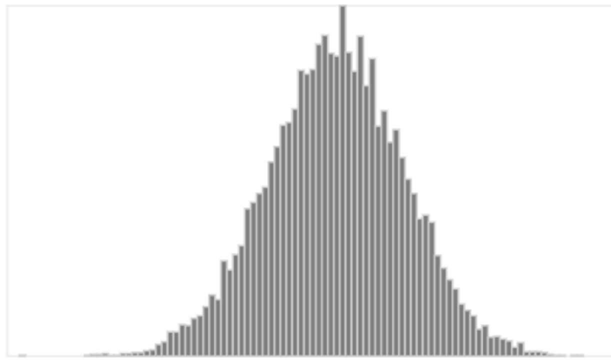
1b) What is printed if you call the function **TestRiddle**, as defined below:

```
function TestRiddle() {
    console.log(mystery(4, "9").puzzle("4", "nine"));
}

function mystery(x, y) {
   let z = 2 * x;
   let w = x + y;
   let obj = {};
   obj.z = x;
   obj.puzzle = function(s, y) {
      if (this.s === undefined) w += ":";
      let v = String.fromCharCode("a".charCodeAt(0) + this.z);
      return w + z + "=" + v + y.substring(1);
   };
   return obj;
}
```

Answer to Problem 1b: ____________________

Problem 2—Simple graphics (15 points)

Present your implementation of the **`Histogram`** factory function, which constructs and returns a **`GCompound`** to display a traditional histogram of final exam scores, as with:



The **`Histogram`** function accepts a **`scores`** array of length of 101, where each index contains the number of students receiving that score. The **`GCompound`** is always 350 pixels wide and 200 pixels high, filled in **`"White"`**, outlined in **`"LightGray"`**. The width of the histogram is filled with up to 101 vertical bars, all of equal width, one for each possible exam score. The reference point should be the **`GCompound`**'s center.

The height of each vertical bar—drawn as a rectangle with a **`"LightGray"`** border and a **`"Gray"`** interior—is proportional to the number of students receiving the represented score. The height of the most common score's vertical bar is scaled to be as tall as the **`GCompound`** itself, and all other vertical bars are scaled proportionately. (When no one receives a particular score, the relevant scores array entry will contain a 0. It's easiest to place a rectangle of height 0.)

Your implementation can rely on the following helper function, which returns the largest value present in the supplied array.

```
/**
 * Function: findMaximum
 * ---------------------
 * Returns the maximum value present in the supplied array of numbers.
 */
function findMaximum(array) {
  let max = array[0];
  for (let i = 0; i < array.length; i++) {
    if (array[i] > max) {
       max = array[i];
    }
  }
  return max;
}
```

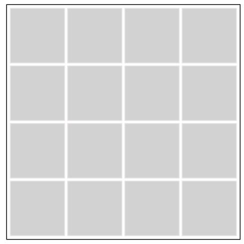
(space for the answer to Problem 2 appears on the next page)

Answer to Problem 2:

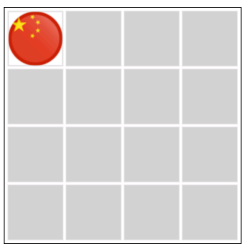
```
const HG_WIDTH = 350;
const HG_HEIGHT = 200;
const HG_BORDER = "LightGray";
const HG_FILL = "White";
const HG_BAR_BORDER = "LightGray";
const HG_BAR_FILL = "Gray";

function Histogram(scores) {
```

Problem 3—Interactive graphics (20 points)

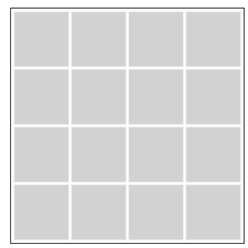
Match the Flags is a memory game where single player is presented with a 4x4 grid of concealed images, as with:



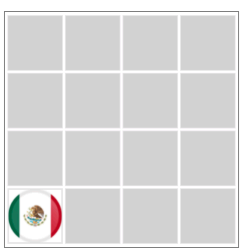
Each image is that of some country's flag, and each image is present not once, but twice. The goal of the game is to click on squares to reveal the flags and match image pairs. Clicking the upper left square and the one below it would reveal two images, as with:

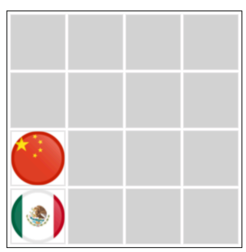

after first click

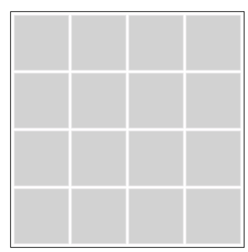

after second click


after one second delay

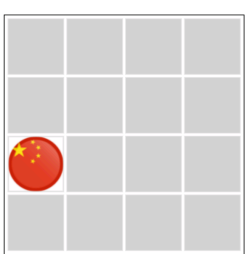
When the two selections mismatch as they do above, you're given **one second** to commit the exposed images to memory before their covers are restored. The next pair of clicks might reveal another mismatch:

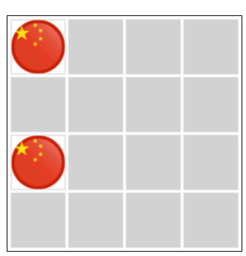

after first click

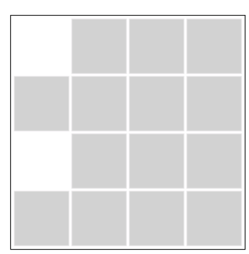

after second click


after one second delay

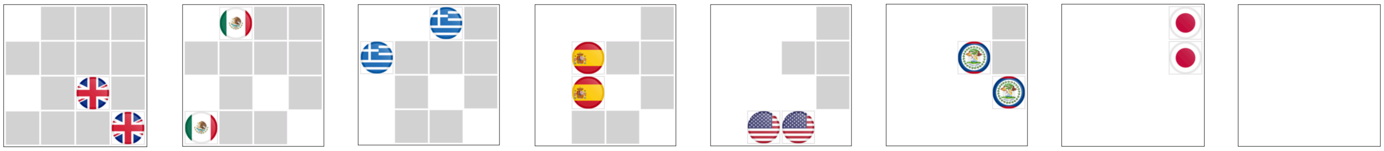
Of course, now we've seen both Chinese flags, so you'd surely click to reveal each of them a second time. Once the two have been exposed, you're given **one second** to enjoy a tiny victory before the two uncovered images are removed from the screen.


after first click


after second click


after one second delay

The player continues to expose pairs of matching flags—presumably with intermittent mismatches I don't bother to show—until all pairs have been matched.



For this problem, you're to complete a partial program to include the mouse event and timer functionality needed to realize the full game experience. The program you're to complete begins like this:

```
function MatchTheFlags() {
   let gw = createBoard(constructBoard());
   // the rest is up to you
```

Rather than overwhelm you with implementations of **constructBoard** and **createBoard**, I'll describe them and further outline how each of sixteen **GCompound**s in the graphics window are structured. You'll then complete the implementation of **MatchTheFlags** to include the mouse event and timeout functionality needed to fully realize the game.

Simply stated, **constructBoard** returns a 4 x 4 array of **GCompound**s, each comprised of an image of a country's flag and a filled gray rectangle above it. Each **GCompound** also includes three additional fields:

- **cover**, which stores a reference to the **GCompound**'s filled rectangle,
- **country**, which stores the name of the country whose flag lay beneath **cover**, and
- **selected**, which is a Boolean initially set to **false**.

(Recall the **GCompound**s in your Enigma assignment did the same thing with their **GLabel**s, so mouse events could change their colors.)

createBoard creates a perfectly sized graphics window, places all 16 **GCompound**s in the correct locations, and then returns the window.

Use the next page to complete the implementation of the **MatchTheFlags** function, subject to the following implementation requirements:

- When the user clicks on a covered flag, toggle the **cover** of the selected **GObject** to be transparent (by invoking **setFilled**) and allowing the flag to show through.
- When the user clicks on a second covered flags, the game should halt for one second, after which mismatched flags should be covered up and matched flags should be fully removed. During this one second delay, user clicks should be ignored.
- When all covered flags have been removed, the game is over. No additional code need be written to address the end of game. User clicks are simply ignored.

Answer to Problem 3:

```
/* Constants */
const DELAY = 1000;

function MatchTheFlags() {
   let gw = createBoard(constructBoard());
   // use the rest of this page to complete the MatchTheFlags function
```

Problem 4—Strings (15 points)

The **Beaufort cipher** is a Caesar cipher-like encryption scheme that accepts a cleartext string—comprised of uppercase letters—and produces an encrypted string of the same length.

This Beaufort cipher relies on a user-supplied encryption key—itself a string of uppercase letters—and the 26 x 26 table drawn on the right. The first row of the table is the uppercase alphabet unmodified, and each row thereafter is a one-letter forward rotation of the row above it.

```
  ABCDEFGHIJKLMNOPQRSTUVWXYZ

A ABCDEFGHIJKLMNOPQRSTUVWXYZ
B BCDEFGHIJKLMNOPQRSTUVWXYZA
C CDEFGHIJKLMNOPQRSTUVWXYZAB
D DEFGHIJKLMNOPQRSTUVWXYZABC
E EFGHIJKLMNOPQRSTUVWXYZABCD
F FGHIJKLMNOPQRSTUVWXYZABCDE
G GHIJKLMNOPQRSTUVWXYZABCDEF
H HIJKLMNOPQRSTUVWXYZABCDEFG
I IJKLMNOPQRSTUVWXYZABCDEFGH
J JKLMNOPQRSTUVWXYZABCDEFGHI
K KLMNOPQRSTUVWXYZABCDEFGHIJ
L LMNOPQRSTUVWXYZABCDEFGHIJK
M MNOPQRSTUVWXYZABCDEFGHIJKL
N NOPQRSTUVWXYZABCDEFGHIJKLM
O OPQRSTUVWXYZABCDEFGHIJKLMN
P PQRSTUVWXYZABCDEFGHIJKLMNO
Q QRSTUVWXYZABCDEFGHIJKLMNOP
R RSTUVWXYZABCDEFGHIJKLMNOPQ
S STUVWXYZABCDEFGHIJKLMNOPQR
T TUVWXYZABCDEFGHIJKLMNOPQRS
U UVWXYZABCDEFGHIJKLMNOPQRST
V VWXYZABCDEFGHIJKLMNOPQRSTU
W WXYZABCDEFGHIJKLMNOPQRSTUV
X XYZABCDEFGHIJKLMNOPQRSTUVW
Y YZABCDEFGHIJKLMNOPQRSTUVWX
Z ZABCDEFGHIJKLMNOPQRSTUVWXY
```

When encrypting cleartext (e.g, **`"THEBRITISHARECOMING"`**) with a key (e.g., **`"ENIGMA"`**), the key is effectively repeated as much as necessary to produce an extended string whose length matches that of the supplied cleartext, so that can be stacked like this:

```
ENIGMAENIGMAENIGMAE
THEBRITISHARECOMING
```

To encrypt the leading T, search for E (that's the letter stacked above it) within T's row of the table. Because E (at position 11) is the 12th character of **`"TUVWXYZABCDEFGHIJKLMNOPQRS"`**, T encrypts to L, which is the 12th character of the uppercase alphabet. Similarly, N appears at position 6 in H's row, which means the H encrypts to the 7th letter of the uppercase alphabet, which is G. The process continues until each cleartext character has been addressed to produce an encrypted string.

```
encrypt("THEBRITISHARECOMING", "ENIGMA") -> "LGEFVSLFQZMJALUUENY"
encrypt("BBBBBBBBBBBBBB", "BCDEFG") -> "ABCDEFABCDEFAB"
encrypt("DEFENDTHEEASTWALLOFTHECASTLE", "FORTIFICATION") -> "CKMPVCPVWPIWUJOGIUAPVWRIWUUK"
```

Using the space on the next page, present your implementation of **`encrypt`**, which codes to the Beaufort cipher algorithm described above. You may assume the supplied cleartext and key parameters consist of just uppercase letters. Your implementation can assume the existence of a constant called **`TABLE`**, which is an array of 26 strings. The 0th string is just the uppercase alphabet, the 1st string is the uppercase alphabet rotated one character so that it leads with B, and so on.

(space for the answer to Problem 4 appears on the next page)

Answer to Problem 4:

```
/* Constants */
const ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
const TABLE = [
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
  "BCDEFGHIJKLMNOPQRSTUVWXYZA",
  "CDEFGHIJKLMNOPQRSTUVWXYZAB",
  // 21 rows omitted for brevity
  "YZABCDEFGHIJKLMNOPQRSTUVWX",
  "ZABCDEFGHIJKLMNOPQRSTUVWXY"
];

function encrypt(cleartext, key) {
```

Problem 5—Arrays (10 points)

Implement the **`leaders`** function to accept an array of integers and return a new array containing just that array's leaders. An element is a ***leader*** if it's strictly greater than all other elements at higher indices. So, for example, the array **`[1, 7, 4, 3, 5, 2]`** has three leaders: 7, 5, and 2. The 7 at index 1 is a leader because it's greater than all numbers at higher indices, the 5 is a leader because it's greater than the only element that comes after it, and the 2 is trivially a leader, because the 2 occupies the last position in the array. Therefore, **`leaders([1, 7, 4, 3, 5, 2])`** would return **`[7, 5, 2]`**.

Use the rest of this page to supply your implementation of **`leaders`**.

(space for the answer to Problem 5 appears below)

```
function leaders(array) {
```

Problem 6—Working with data structures (15 points)

Because I'm turning 50 years old on December 28th, I've decided to throw the biggest of big dinner parties and cook everything myself. I've decided what I'm cooking, and I've even converged on the collection of recipes I'll prepare for the event. As your birthday present to me, you've decided to write a program to scale up recipes so I know precisely how much of each ingredient to buy at the supermarket.

Take for example this lovely soup dish I found online, coincidentally expressed in JSON:

```
let soup = {
   name: "Charred Cauliflower Stew",
   servings: 4,
   ingredients: [
      { amount: 2, unit: "head", name: "cauliflower"},
      { amount: 2, unit: "teaspoon", name: "olive oil"},
         ... many more ingredients omitted
      { amount: 1, unit: "cup", name: "watercress"},
   ],
   instructions: [
      "Heat the grill to high and lightly oil the grates.",
      "Toss cauliflower florets with half of the olive oil.",
      ... many more instructions omitted
      "Garnish with remaining watercress and serve."
   ]
};
```

The problem? If I need to cook for 16 people instead of 4, I'd prefer the recipe be structured as follows:

```
{
   name: "Charred Cauliflower Stew",
   servings: 16,
   ingredients: [
      { amount: 8, unit: "head", name: "cauliflower"},
      { amount: 1.33, unit: "ounce", name: "olive oil"},
         ... many more ingredients omitted
      { amount: 1, unit: "quart", name: "watercress"},
   ],
   instructions: [
      "Heat the grill to high and lightly oil the grates.",
      "Toss cauliflower florets with half of the olive oil.",
      ... many more instructions omitted
      "Garnish with remaining watercress and serve."
   ]
};
```

Since the recipe was quadrupled, 2 teaspoons became 8 teaspoons and 1 cup became 4 cups. But those quantities are better expressed in larger units of measure, which is why 8 teaspoons isn't 8 teaspoons or even 2.66 tablespoons, but rather 1.33 ounces. When a quantity can be expressed in terms of 1.0 or more of the next larger unit of measure, it is.

If instead of a party of 16 I throw a party for 1600 (**`#lifegoals`**), I'll need a drastically reformed ingredient list for cauliflower soup, as with:

```
{
   name: "Charred Cauliflower Stew",
   servings: 1600,
   ingredients: [
      { amount: 800, unit: "head", name: "cauliflower"},
      { amount: 1.042, unit: "gallon", name: "olive oil"},
         ... many more ingredients omitted
      { amount: 3.125, unit: "bushel", name: "watercress"},
   ],
   instructions: [
      "Heat the grill to high and lightly oil the grates.",
      "Toss cauliflower florets with half of the olive oil.",
      ... many more instructions omitted
      "Garnish with remaining watercress and serve."
   ]
};
```

These conversions are possible, provided we have an ordered list of conversions for each of the standard units of measures. Fortunately, we do, because the internet has everything.

```
const CONVERSIONS = [
   { unit: "dram", amount: 4/3 },
   { unit: "teaspoon", amount: 3 },
   { unit: "tablespoon", amount: 2 },
   { unit: "ounce", amount: 8 },
   { unit: "cup", amount: 2 },
   { unit: "pint", amount: 2 },
   { unit: "quart", amount: 4 },
   { unit: "gallon", amount: 2 },
   { unit: "peck", amount: 4 },
   { unit: "bushel", amount: 55/7 },
   { unit: "barrel", amount: 6000},
   { unit: "acre" } // no larger unit of measure, so no amount field
];
```

Each object in the array constant stores the number of a particular unit needed to make up exactly one of the next larger unit. Therefore, 3 teaspoons make up one tablespoon, 2 tablespoons make up one ounce, 8 ounces make up one cup, and so forth.

Using the space on the next page, implement the **`scale`** function, which accepts a recipe object (structured as those you've seen above) and the desired number of servings, and modifies the recipe object with the proper ingredient lists and amounts needed to serve those many people. If an ingredient's unit of measure appears in **`CONVERSIONS`**, the quantity and unit of measure should be normalized as they have been in the above examples. You must allow for the possibility the supplied recipe needs to be scaled up or down. Hint: convert **all** quantities to drams and then normalize up to the most sensible unit of measure.

```
const CONVERSIONS = [
   { unit: "dram", amount: 4/3 }, { unit: "teaspoon", amount: 3 },
   { unit: "tablespoon", amount: 2 }, { unit: "ounce", amount: 8 },
   { unit: "cup", amount: 2 }, { unit: "pint", amount: 2 },
   { unit: "quart", amount: 4 }, { unit: "gallon", amount: 2 },
   { unit: "peck", amount: 4 }, { unit: "bushel", amount: 55/7 },
   { unit: "barrel", amount: 6000}, { unit: "acre" }
];

function scale(recipe, servings) {
```

Problem 7— Reading data structures from embedded XML (15 points)

In World War II, the German Enigma operators consulted a codebook to set up the machine for each day's transmission. The complete settings consisted of the following pieces of information (of which you implemented only the second in Assignment 5):

- The rotor order: The wartime Enigma machines came equipped with a box of five rotors, which could be inserted into the machine in any order. The codebook told the operator which rotors to choose for the three rotors. For example, the rotor order 513 asks the operator to use stock rotor 5 as the slow rotor, stock rotor 1 as the medium rotor, and stock rotor 3 as the fast rotor.
- The rotor setting: The rotor setting was a three-letter code showing what letters should be set on the three rotors. For example, several examples in the assignment handout used JLY.
- The Stecker pairings: During the war, the German military also added a Steckerbrett—that's German for plug board—to the front of the apparatus, which would swap pairs of letters at the beginning and end of the encryption process. Each Enigma machine used up to ten pairs of letters as Stecker pairings.

Imagine a new version of Assignment 5 is being constructed to support all of the above, and that all of that information is encoded using XML in the primary **`index.html`** file used for the assignment. Your job here is to extract and parse XML fragments to access these codebook values for every day in a year.

The XML fragments for the first two days of March 1944 might look like this:

```
<div id="codebook" style="display:none">
   ... entry tags for days preceding March 1st, 1944
   <entry date="1944-03-01" order="514" settings="JLY">
      <pairing value="AC"/>
      <pairing value="LS"/>
      <pairing value="BQ"/>
      <pairing value="HP"/>
   </entry>
   <entry date="1944-03-02" order="132" settings="MME">
      <pairing value="FG"/>
      <pairing value="NV"/>
      <pairing value="EY"/>
      <pairing value="RZ"/>
      <pairing value="LW"/>
      <pairing value="CI"/>
      <pairing value="DK"/>
   </entry>
   ... entry tags for days after March 2nd, 1944
</div>
```

The first of the two entries shown tells the Enigma simulation that on March 1st, 1944, rotors **should** be inserted in the order 514, the rotors should be initially set to J, L, and Y, and that four wires should be added to the plug board connecting A to C, L to S, B to Q, and H to P. The XML entry for March 2nd, 1944 is similar, except that seven (not four) wires should be added to the plug board.

For this problem, your job is to write the function

```
function buildCodebook()
```

which parses XML data structured like that on the preceding page and returns a map linking dates to aggregates describing machine configurations. The components of each aggregate are:

- An **`order`** field, which is an integer
- A **`setting`** field, which is a three-letter string
- A **`pairings`** field, which is an array of up to ten two-letter strings

Calling **`buildCodebook`** to process the XML data described on the preceding page should construct and return the following data structure:

```
{
   ... additional key value pairs for other days
   "1944-03-01": {
      order: 514,
      setting: "JPY",
      pairings: ["AC", "LS", "BQ", "HP"]
   },
   "1944-03-01":{
      order: 132,
      setting: "MME",
      pairings: ["FG", "NV", "EY", "RZ", "LW", "CI", "DK"]
   },
   ... additional key value pairs for other days
}
```

While answering this question, you should keep the following in mind:

- All you have to do is read the data into the internal structure. Any code that uses the codebook data structure is the responsibility of your clients.
- You do not need to do any error checking. Assume the XML segment within the HTML file is properly structured and consistent with the sample XML on the prior page.

(space for the answer to Problem 7 appears on the next page)

Answer to Problem 7:

```
function buildCodebook() {
```

Answer to Problem 7 (continued)