Jerry Cain
CS106AX

Handout #16
November 1, 2023

Midterm 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 70. 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 50 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: Simple JavaScript expressions, statements, and methods (10 points)

(1a) [3 points] Compute the value of each of the following JavaScript expressions:

```
10 % 5 + 10 % 15                 ______________

5 > 3 || 10 / 0 === 0            ______________

10 + 15 + "XYZ" + 7 * 8          ______________
```

(1b) [3 points] Assume that the function **`crunch`** has been defined as given below:

```
function crunch(n) {
   while (n >= 10) {
      let k = 1;
      while (n > 0) {
         k *= n % 10;
         n = Math.floor(n / 10);
      }
      n = k;
   }
   return n;
}
```

What is the value of **`crunch(112911)`**?

(1c) [4 points] What output is printed by a call to **`whitney()`**?

```
function whitney() {
   let doris = "madonna";
   let pink = function(x, y, s) {
      return s.substring(x) + doris.substring(x, y);
   };
   doris = vocalist(pink, doris.indexOf("a"), doris.lastIndexOf("on"));
   console.log(doris);
}

function vocalist(fn, x, y) {
   let dolly = fn(x, y, "beyonce");
   dolly += "789".charCodeAt(2) - "123".charCodeAt(0);
   return dolly.toUpperCase();
}
```

Problem 2: Using graphics and animation (15 points)

Implement a graphical program that populates an initially empty graphics window with circles, one every 200 milliseconds, without end. Each circle should be 20 pixels in diameter, filled, and both the border and fill color should initially be `"Green"`. The position of each circle should be random as well, though you should ensure the entirely of the circle fits within the graphics window. Once placed, the circle should be removed from the graphics screen after three full seconds (or rather, 3000 milliseconds) if it hasn't already been removed by a pair of mouse clicks, as described below.

The program should also respond to mouse clicks, and whenever you click on a green circle, its color should change to `"Yellow"`. Whenever you click on a yellow circle, the circle should be removed from the graphics window. If there are multiple circles stacked at a particular location, your click need only impact one of them (and most likely, the one most recently added to the graphics window).

A few things:

- You already know that all objects respond `setColor`, which allows you to specify what color the border (and unless specified to be different, the interior) should be. There is also the `getColor` method, that returns the color as a string.
- `gw.remove(object)` removes the identified object if it appears in the graphics window. If object is no longer in the graphics window but `gw.remove(object)` is called anyway, the call to `gw.remove(object)` has no effect.
- Recall that `setInterval(func, period)` schedules `func` to execute every period milliseconds but that `setTimeout(func, delay)`, on the other hand, schedules `func` to execute just once, `delay` milliseconds in the future.

Use the space on the next page to present your implementation.

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

```
/* Constants */
const GWINDOW_WIDTH = 500;   // in pixels
const GWINDOW_HEIGHT = 300;  // in pixels
const CIRCLE_RADIUS = 10;    // in pixels
const STEP_TIME = 200;       // in milliseconds
const MAX_LIFETIME = 3000;   // in milliseconds

/* Derived Constants */
const CIRCLE_DIAMETER = 2 * CIRCLE_RADIUS;

function RandomCircles() {
   // present your implementation in the space below
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
```

Problem 3: Strings (15 points)

Implement the **`rotate`** function, which accepts a string (one you can assume is comprised of lowercase letters and nothing else) and returns a new string that's the same as the original, save for the fact that all the vowels have been *rotated*. Restated, **`rotate`** returns a new string which is a replica of the original, except that the first vowel in the new string should be the second vowel of the original, the second vowel in the new string should be the third vowel of the original, and so forth. For completion, the last vowel of the new string should be equal to the first vowel of the original. So, for example, the vowel rotation of the word **`"illuminate"`** would be **`"ullimaneti"`**, where you see that the i, u, i, a, and e have been replaced by u, i, a, e, and i, respectively. If there's only one instance of a vowel or there aren't any vowels at all, then rotate should return just the same word.

Here are some examples of how **`rotate`** operates:

```
         rotate("abbey") ⇒ "ebbay"
     rotate("seriously") ⇒ "sirouesly"
 rotate("embellishment") ⇒ "embilleshment"
    rotate("antithesis") ⇒ "intethisas"
         rotate("think") ⇒ "think"
           rotate("xyz") ⇒ "xyz"
              rotate("") ⇒ ""
```

Use the next page to present your implementation of **`rotate`**. Your solution should rely on the supplied **`findFirstVowel`** function, and you're unlikely to need any JavaScript String methods other than **`substring`**.

(space for the answer to problem #3 appears on the next page)

```
function isVowel(ch) {
   return "aeiou".indexOf(ch) !== -1;
}

function findFirstVowel(str, start) {
   if (start === undefined) start = 0;
   for (let i = start; i < str.length; i++) {
         if (isVowel(str.charAt(i))) {
               return i;
         }
   }

   return -1;
}

function rotate(str) {
  // present your implementation in the space below
```

Problem 4: Arrays (15 points)

The **look-and-say** sequence is the sequence of numbers beginning as:

```
1
11
21
1211
111221
312211
13112221
```

Each number in the sequence is generated by "reciting" its predecessor, and then capturing what was said in a number format that should be clear from the example below.

Reciting 111221 out loud, you'd look and say: 3 ones, followed by 2 twos, followed by 1 one, or 312211. Reading 312211 aloud, you'd say: 1 three, 1 one, 2 twos, 2 ones, or 13112221.

look-and-say arrays are similar, but numbers are instead expressed as arrays of isolated digits, as with [3, 1, 2, 2, 1, 1]. Reading the array aloud, you'd say 1 three, 1 one, 2 twos, 2 ones, which we'd captured in array form as [1, 3, 1, 1, 2, 2, 2, 1].

For this problem, you're to write the **`lookandsay`** function, which accepts an array of single digit numbers and generates its successor according to the look-and-say rules outlined here. For example:

```
      lookandsay([3, 1, 2, 2, 1, 1]) returns [1, 3, 1, 1, 2, 2, 2, 1]
lookandsay([1, 3, 1, 1, 2, 2, 1, 1]) returns [1, 1, 1, 3, 2, 1, 2, 2, 2, 1]
```

Use the next page to present your implementation. You're unlikely to need any JSArray directives other than the **`.length`** property and the **`push`** method.

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

```
function lookandsay(digits) {
  // present your implementation in the space below
```

Problem 5: Working with data structures (15 points)

By now, you're all quite familiar with the game of Wordle, even if you'd somehow missed news of it prior to CS106AX's Assignment 3. Of course, the goal is to uncover a secret word via a series of six or fewer educated guesses. With each guess, the game identifies correctly placed letters by coloring them green and further identifies correctly guessed but incorrectly placed letters by coloring them yellow.

A JavaScript object modeling a large subset of all users and their game play for any given game might look like this:

```
let today = {
    secret: "phony",
    users: [ {
               time: 74.1,
               guesses: ["slate", "pound", "prong", "phony"]
             }, {
               time: 124.3,
               guesses: ["heart", "fishy", "money", "peony", "phony"]
             },
             … many additional users
                {
               time: 283.5,
               guesses: ["blast", "grime", "pluck", "poppy", "phone", "phony"]
             },
           ]
};
```

Our object contains the secret word and a large array of user objects, where each object stores the sequence of guesses that led them to a win or loss alongside the number of seconds it took that same user to enter their final guess. As you can see, the 0th user took 74.1 seconds to work through their four guesses to ultimately match the secret word, whereas the last user took 283.5 seconds to work through all six guesses afforded them to win. Of course, it's possible the user doesn't win the game at all and that none of the words in the array match the secret one.

For this problem, you're to implement a **`statistics`** function, which accepts a JavaScript object like that structured above and returns a separate JavaScript object with two fields called **`average`** and **`histogram`**. The **`average`** field should store the average number of seconds it took to win the game (being careful to exclude those that didn't win from the calculation), and the **`histogram`** field should store an array of length 7, where index *0* stores the number of users who failed to guess the secret word, but otherwise , index *k* stores the number of users who took precisely *k* guesses to win (e.g., index 3 should store the number of users who matched the secret word on the 3rd guess).

Place your implementation of **`statistics`** on the next page.

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

```
function statistics(info) {
   // present your implementation in the space below
```