Jerry Cain
CS 106AX

Handout #10
October 23, 2023

Practice Midterm Examination

Midterm exams: Wednesday, November 1, 3:30–5:30 P.M., 200-002
Wednesday, November 1, 7:00–9:00 P.M., 370-370

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the midterm examination a week from Wednesday. A solution set to both practice examinations will be posted on Friday.

Recall that our midterm is closed computer but open book, and you may use any notes or materials from the class.

Time and place of the exam

The midterm exam is scheduled at two different times to accommodate those of you who have scheduling constraints. You're permitted to take the exam at either time and you needn't give advance notice of which exam you plan to take. If you cannot take the exam at either of the two scheduled times or if you need special accommodations, please send an email message to jerry@cs.stanford.edu stating the following:

- The reason you cannot take the exam at either of the scheduled times and/or the details of the special accommodations you require.
- A list of two-hour periods (or longer if you have OAE accommodations) on Wednesday or Thursday during which you could take the exam. These time blocks must be during the regular working day and must therefore start between 8:30 and end by 5:00.

To schedule an alternate exam, we must receive an email message from you by 5:00 P.M. on Friday, October 7th. Instructions for taking the alternate midterm will be sent to you via email by Monday, October 30th.

Coverage

The exam covers the material presented in class through Monday, October 23rd, which means that you are responsible for the material in Chapters 1 through 8 of the textbook. You're also expected to understand and have implemented Assignments 1 through 4.

Note: To conserve trees, we have cut back on answer space for the practice midterm. The actual exam will have much more room for your answers and for any scratch work.

General instructions

Answer each of the five questions included in the exam. 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 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.

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

Unless otherwise indicated as part of the instructions for a specific problem, comments are not required. 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 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.

Problem 1: Simple JavaScript expressions, statements, and methods (10 points)

(1a) Compute the value of each of the following JavaScript expressions:

```
5 % 4 - 4 % 5                    _____
```

```
7 < 9 - 5 && 3 % 0 === 3         _____
```

```
"B" + 3 * 4                      _____
```

(1b) Assume that the function **`mystery`** has been defined as given below:

```
function mystery(str) {
   let result = "";
   let len = str.length;
   let j = 0;
   let k = 9;
   while (j < k) {
      if (j < 4) {
         result += str.charAt(k % len);
      }
      if (Math.floor(j / 2) !== 1) {
         result += str.charAt(j % len);
      }
      j++;
      k--;
   }
   return result;
}
```

What is the value of

```
mystery("abcdefg")
```

(1c) What output is printed by a call to **`problem1c()`**?

```
/**
 * File: Problem1c.js
 * ------------------
 * This program doesn't do anything useful and exists only to
 * exercise your understanding of parameters and string methods.
 */

function problem1c() {
   let s1 = "To err";
   let s2 = "is human!";
   s1 = forgive(s1, s2);
   console.log(s1 + " " + s2);
}

function forgive(me, you) {
   let heart = me.substring(0, you.length - me.length);
   you = "" + you.charAt(me.length);
   let amount = heart.length;
   me = me.substring(amount + 2) + me.charAt(amount);
   heart += understanding(you, 2) + you + me;
   return heart;
}

function understanding(you, num) {
   return String.fromCharCode(you.charCodeAt(0) + num);
}
```

Problem 2: Using graphics and animation (15 points)

Write a graphical program that does the following:

1. Creates the following cross as a `GCompound` containing two filled rectangles:

   The color of the cross should be red, as in the emblem of the International Red Cross (it actually is red in the diagram, but that's hard to see on the printed copies, which are of course in black and white). The horizontal rectangle should be 60×20 pixels in size and the vertical one should be 20×60. They cross at their centers.
2. Adds the cross to the graphics window so that it appears at the center.
3. Moves the cross at a speed of 2 pixels every 20 milliseconds in a random direction, which is specified as a random real number between 0 and 360 degrees.
4. Every time you click the mouse inside the cross, its direction changes to some new random direction—again chosen as a random real number between 0 and 360 degrees—but its velocity remains the same. Clicks outside the cross have no effect.

If you were actually to write such a program, you would presumably supply some means of making it stop, such as when the cross moves off the screen. For this problem, just have the program run continuously without worrying about objects moving off screen or how to stop the timer.

Problem 3: Strings (15 points)

A ***spoonerism*** is a phrase in which the leading consonant substrings of the first and last words are inadvertently swapped, generally to comic effect. Some examples of spoonerisms include the following phrases and their spoonerized counterparts (the consonant strings that get swapped are underlined):

crushing blow → *blushing crow*
sons of toil → *tons of soil*
pack of lies → *lack of pies*
jelly beans → *belly jeans*
flutter by → *butter fly*

In this problem, your job is to write a function

```
function spoonerize(phrase)
```

that takes a multiword phrase as its argument and returns its spoonerized equivalent. For example, you should be able to use your function to duplicate the following console session in which all the examples come from Shel Silverstein's spoonerism-filled children's book *Runny Babbit:*

```
JavaScript Console
> spoonerize("bunny rabbit")
runny babbit
> spoonerize("silly book")
billy sook
> spoonerize("take a shower")
shake a tower
> spoonerize("wash the dishes")
dash the wishes
>
```

In this problem, you are not responsible for any error-checking. You may assume that the phrase passed to **`spoonerize`** contains nothing but lowercase letters along with spaces to separate the words. You may also assume that the phrase contains at least two words, that there are no extra spaces, and that each word contains at least one vowel. What your method needs to do is extract the initial consonant substrings from the first and last words and then exchange those strings, leaving the rest of the phrase alone.

Hint: Remember that you can use methods from the book. The **`findFirstVowel`** and **`isEnglishVowel`** methods from the Pig Latin example will certainly come in handy.

Problem 4: Arrays (15 points)

Implement the **`leaders`** function to accept an array of integers and return a new array containing just that array's leaders, preserving order. 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]`**.

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

Problem 5: Working with data structures (15 points)

Adventure was not the first widely played computer game in which an adventurer wandered in an underground cave. As far as we know, that honor belongs to the game "Hunt the Wumpus", which was developed by Gregory Yob in 1972.

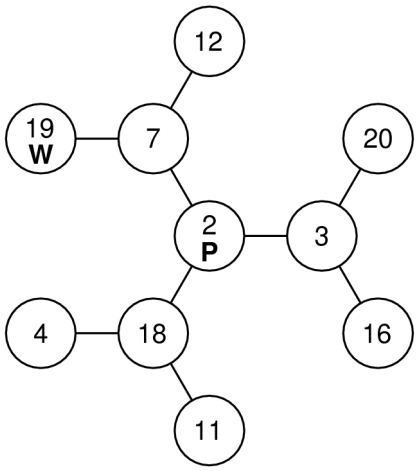
In the game, the wumpus is a fearsome beast that lives in an underground cave composed of 20 rooms, each of which is numbered between 1 and 20. Each of the twenty rooms has connections to three other rooms, represented as an array—one element per room—of three-element arrays containing the numbers of the connecting rooms. (Because the room numbers don't start with 0, the data structure puts a **`null`** in element 0 of the outer array, which is never used.) In addition to the connections, the data structure for the wumpus game also keeps track of the room number occupied by the player and the room number in which the wumpus resides.

In an actual implementation of the wumpus game, the information in this data structure would be generated randomly. For this problem, which focuses on whether you can work with data structures that have already been initialized, you can imagine that the data structure has been initialized using the JSON notation shown in Figure 1.

Figure 1. JSON representation of the data structure for the wumpus cave

```
const WUMPUS_CAVE = {
   playerLocation: 2,      // The player is in room 2
   wumpusLocation: 19,     // The wumpus is in room 19
   connections: [
      null,          // room 0 is not used
      [6, 14, 16],   // room 1 connects to rooms 6, 14, and 16
      [3, 7, 18],    // room 2 connects to rooms 3, 7, and 18
      [2, 16, 20],   // room 3 connects to rooms 2, 16, and 20
      [6, 18, 19],   // and so forth
      [8, 9, 11],
      [1, 4, 15],
      [2, 12, 19],
      [5, 10, 13],
      [5, 11, 17],
      [8, 14, 16],
      [5, 9, 18],
      [7, 14, 16],
      [8, 15, 20],
      [1, 10, 12],
      [6, 12, 13],
      [1, 3, 10],
      [9, 19, 20],
      [2, 4, 11],
      [4, 17, 17],
      [3, 13, 17]
   ]
};
```

Looking at the data structure allows you to diagram the rooms in the cave. Here is a piece of the cave map centered on the location of the player in room 2:



The player is in room 2, which has connections to rooms 3, 7, and 18. Similarly room 7 has connections to rooms 2, 12, and 19, which is where the wumpus is lurking. The other connections from rooms 4, 11, 16, 20, 12, and 19 are not shown.

It was usually possible to avoid the wumpus because, like the Sasquatch in the Sherman Alexie poem, the wumpus was so stinky that the player could smell the wumpus from up to two rooms away. Thus, in the diagram above, the player can smell the wumpus. If, however, the wumpus were to wake up and move to a room beyond the boundaries of this diagram, the scent of the wumpus would disappear.

Write a predicate function **`playerSmellsWumpus`**, which takes the entire data structure as an argument and returns **`true`** if the player smells the wumpus and **`false`** otherwise. Thus, calling

```
playerSmellsWumpus(WUMPUS_CAVE)
```

would return **`true`**. The function would also return **`true`** if the wumpus were in rooms 3, 7, or 18, which are one room away from the player. If, however, the wumpus were in one of the rooms not shown on this map, **`playerSmellsWumpus`** would return **`false`**.