Jerry Cain
CS 106AX

Handout #14
October 24th, 2026

Practice Midterm Examination

Midterm exams: Tuesday, October 27th, 3:30–5:30 P.M., ART 360
Tuesday, October 27th, 7:00–9:00 P.M., 370-370

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

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

```
3 + 2 * 2 - 15 % 5 * 100
```
_____

```
"B" === "b" || "H" < "GGG"
```
_____

```
20 + 7 + "1" + 8 + 4 * 7
```
_____

(1b) Assume that the method **`riddle`** has been defined as follows:

```
function riddle(str) {
  let result = "";
  for (let i = 0; i < str.length; i++) {
     if (i === str.lastIndexOf(str.charAt(i))) {
        result += str.charAt(i);
        str = str.substring(i + 1);
        i = -1;
     }
  }
  return result;
}
```

What is the value returned by **`riddle("mirrormirror")`**?

(1c) What output is printed by the following **Problem1c** program?

```
function Problem1c() {
   let day = "halloween";
   let fn = function(x, y, z) {
      return z.substring(x, y) + day.substring(y);
   };
   day = spooky(fn, day.indexOf("a"), day.indexOf("o"));
   day.toLowerCase();
   console.log(day);
}

function spooky(f, x, y) {
   let ghost = f(x, y, "nightmare");
   ghost += "xyz".charCodeAt(1) - "a".charCodeAt(0);
   return ghost.toUpperCase();
}
```

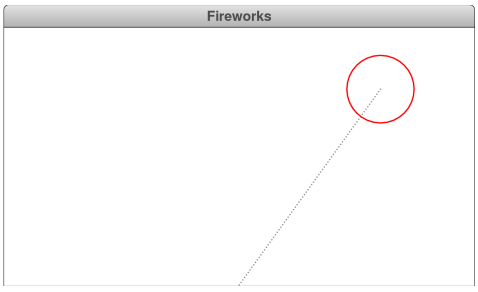
Problem 2: Using graphics and animation (15 points)

When Marissa Mayer (former CEO of Yahoo! Inc.) took CS 106A, her entry in the Graphics Contest was a screensaver program that simulated fireworks. Your task in this problem is to implement an exam-sized subset of her contest-sized application.

Your program should execute the following steps:

1. Create a tiny dot (an unfilled circle whose width and height are both one pixel) at the point that lies at the center of the bottom of the window and color it using a randomly chosen color.
2. Choose a random point somewhere in the top half of the window.
3. Animate the motion of the dot so that it moves to the point you chose in step 2. The dot should move so that gets to its destination in **FLIGHT_TIME** milliseconds.
4. Once the dot reaches its destination, you should change the behavior of the animation so that the circle radius increases by **DELTA_RADIUS** pixels for **EXPANSION_TIME** milliseconds.

An animation with random behavior is difficult to represent on the printed page, but the final image on the window will look something like this (the dotted line shows the flight path of the dot, although this line would not appear on the window):




```
function Fireworks() {
```

Problem 3: Strings (15 points)

A ***portmanteau*** is a linguistic compression of two words into one. For example, the word *smog* is a portmanteau of *smoke* and *fog*. Similarly, the vegetarian Thanksgiving alternative called *tofurkey* is a portmanteau of *tofu* and *turkey*.

For this problem, we've restricted the rules for creating a portmanteau so that they apply only to words that contain a common vowel. Under this rule, a portmanteau is formed by searching for the leftmost vowel in the first word that also appears in the second word, as in the *o* in *smoke* and *fog* or the *u* in *tofu* and *turkey*. Once a vowel common to both is identified, the compression is generated by taking everything in the first word up to and including the first occurrence of the shared vowel and prepending it to everything in the second word appearing beyond its first instance of that same vowel. That means `"smog"` is really `"smo"` followed by `"g"`, and `"tofurkey"` is really `"tofu"` followed by `"rkey"`.

Write a function

```
function portmanteau(word1, word2)
```

that creates the portmanteau of `word1` and `word2` using this rule. If the two words do not contain a common vowel, your function should return the constant `null` to show that no portmanteau is possible under this rule. You may assume both of the supplied words are all lowercase. And you're more than welcome to decompose, as our own solution relied on two helper functions (one of which we've already seen in lecture).

Problem 4: Arrays (15 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`.

```
function dedupe(array) {
```

Problem 5: Working with data structures (15 points)

Although it is hard to imagine now, Facebook's IPO in 2012 didn't go as well as predicted, and the Morgan Stanley brokerage that handled the offering was forced to make restitution to some clients, primarily for late trades. Suppose, for example, that a client ordered a sale at 11:28am on May 18, when Facebook was selling at $40.00 a share. Given the many delays on that day, Morgan Stanley might not have been able to execute the sell order until 3:58pm, when Facebook shares had dropped to $38.07. That client therefore lost $1.93 per share, which adds up quickly if the trade involved a large block of shares.

Suppose that Morgan Stanley has hired you to write a simple application to calculate refunds due to its customers. You have access to a data structure that contains the

complete history of the share price for Facebook in the early days of trading. The data structure is an array of aggregates, each of which has three fields: a **`date`** field containing the date as a string (as in **`"5/18/2012"`** for May 18, 2012), a **`time`** field indicating the time as a string (as in **`"11:30am"`**), and a **`price`** field as a number. A few entries in that array look like this when represented in JSON form:

```
const FB_SHARE_PRICE_DATA = [
   { date:"5/18/2012", time:"11:30am", price:42.0000 },
   { date:"5/18/2012", time:"11:31am", price:42.0125 },
   { date:"5/18/2012", time:"11:32am", price:42.0250 },
   { date:"5/18/2012", time:"11:33am", price:42.0250 },
   { date:"5/18/2012", time:"11:34am", price:40.9474 },
   { date:"5/18/2012", time:"11:35am", price:40.8425 },
   { date:"5/18/2012", time:"11:36am", price:40.1500 },
   { date:"5/18/2012", time:"11:37am", price:40.0367 },
   { date:"5/18/2012", time:"11:38am", price:40.0000 },
       . . . more entries for May 18 . . .
   { date:"5/18/2012", time:"3:55pm", price:38.0685 },
   { date:"5/18/2012", time:"3:56pm", price:38.1050 },
   { date:"5/18/2012", time:"3:57pm", price:38.0997 },
   { date:"5/18/2012", time:"3:58pm", price:38.0700 },
   { date:"5/18/2012", time:"3:59pm", price:38.2599 },
   { date:"5/18/2012", time:"4:00pm", price:38.2699 },
];
```

Write a function

```
function facebookRefund(nShares, date, timeOrdered, timeExecuted)
```

that uses the data in **`FB_SHARE_PRICE_DATA`** to compute the refund due to a customer who tried to sell **`nShares`** of Facebook stock if the order was made at **`timeOrdered`** on the specified date but the order was not completed until **`timeExecuted`**.

As an example, suppose that a client tried to sell 1000 Facebook shares at 11:38am on May 18, but Morgan Stanley was unable to complete the transaction until 3:58pm. You can compute the necessary refund by calling

```
facebookRefund(1000, "5/18/2012", "11:38am", "3:58pm")
```

The implementation has to look through the **`FB_SHARE_PRICE_DATA`** array to find the price of Facebook stock at the two specified times. From the table on the preceding page, you can see that Facebook stock was selling for $40.00 per share at 11:38am but had dropped to $38.07 per share by 3:58pm. Morgan Stanley's delay therefore cost the client $1.93 per share. Since the client was selling 1000 shares, the total refund is $1,930.00, which is the value that **`facebookRefund`** should return. The value of **`facebookRefund`** should never be negative. If the client profited from the delay, **`facebookRefund`** should simply return 0. For simplicity, you may assume that **`FB_SHARE_PRICE_DATA`** contains the date and two times passed in.