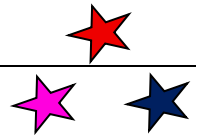


Ben's Practice Midterm Exam

Written by Ben Yan



Here's a practice midterm that I wrote myself, as a terrarium for all the colorful & chaotic ideas swirling in my brain, and with the intention of helping you rehearse and rehydrate your problem-solving skills before the real exam. As such, these problems are designed to be comparable in raw content, difficulty, ~~creativity~~ creativity, and Shakespearean tragedy to Jerry's great and celebrated literature of CS exams.

I hope you enjoy them!

We'll also be performing a set list of these problems during the review session, so no worries at all if you don't get around to this. Please do prioritize, in your own study, the exams written by Jerry himself! If you elect to work through these, you're more than welcome to work with study buddies. You can find more practice problems in the ✨ **Ben's Extra Midterm Problems** ✨ handout that accompanies this one.

Problems (70 Points Total)

1. Tracing That Dream (10 points)
2. Sparks Fly (15 points)
3. Shape-Shifting Cipher (15 points)
4. Paperless Rings (15 points)
5. Spotify Wrapped (15 points)

Topics

JavaScript Expressions, Tracing
 JavaScript Graphics and Animation
 JavaScript Strings
 JavaScript Arrays
 JavaScript Objects, Data Structures

In case you didn't get the memo from the table of contents above, midterm is all JavaScript, no Python ☺.

Thanks everyone – you're going to do wonderful on your first CS midterm here at Stanford!!

Note: No artificial intelligence was used in the writing of this practice exam. The only person I ripped off is Jerry Cain ☺. All characters appearing in this work are fictitious. Any resemblance to real people is purely coincidental.

First and Last Name: _____

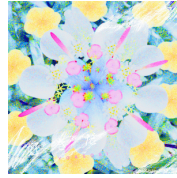
SUNet ID (your @stanford.edu address): _____

Section Leader: _____

you will break your SL's heart if you don't remember them lol



Problem 1: Tracing that Dream (JavaScript Expressions and Tracing)



1. [3 points] Compute the value of each of the following JavaScript expressions.

`2026 + 3 % 2 * 2 - 2020`

`"IDOL" === "idol" || "YUUSHA" < "SUKI" + "DA"`

`6 + "SEIS" + 1 + 2 + 4 + "SIETE"`

2. [3 points] Assume the function **LoveLetter**, which accepts arrays **truth** and **obfuscation** as input, is defined as follows. Note that it returns a new array of identical length to **truth**.

```
function LoveLetter(truth, obfuscation) {
  let encrypted = [];
  let ambiguity = obfuscation.length;

  for (let i = 0; i < truth.length; i++) {
    let mySecret = truth[i];
    let feelings = i % ambiguity;

    let myWords = mySecret + obfuscation[feelings];
    if (myWords >= 26) myWords = myWords - 26;

    encrypted.push(myWords);
  }
  return encrypted;
}
```

What does the call **LoveLetter**([2, 3, 5, 7, 11, 13], [17, 19, 23]) return? Work through this function trace carefully, and indicate the value of each of the elements in the returned array below.

0	1	2	3	4	5

3. [4 points] What output is produced by a call to the function `DomainExpansion()`?

```
function DomainExpansion() {
  let cursed = "energy";
  let technique = function(blue, red, purple) {
    return cursed.substring(blue, red) + purple.substring(red);
  }
  let sorcery = Battle(technique, cursed.lastIndexOf("e"),
                      cursed.indexOf("e"));
  sorcery.toLowerCase();
  console.log("This truly was our " + sorcery);
}

function Battle(cursedTech, positive, negative) {
  let culling = cursedTech(negative, positive, "jujutsu");
  let games = "shibuya".charCodeAt(5) - "arc".charCodeAt(0);
  let tokyo = "tombsOfStar".charAt(3); // not the same as charCodeAt!
  return tokyo.toUpperCase() + culling + games;
}
```

Problem 2: Sparks Fly (Graphics and Animation, 15 points)

"Cause I see sparks fly, whenever you smile" – Taylor Swift (song: Sparks Fly, album: Speak Now)

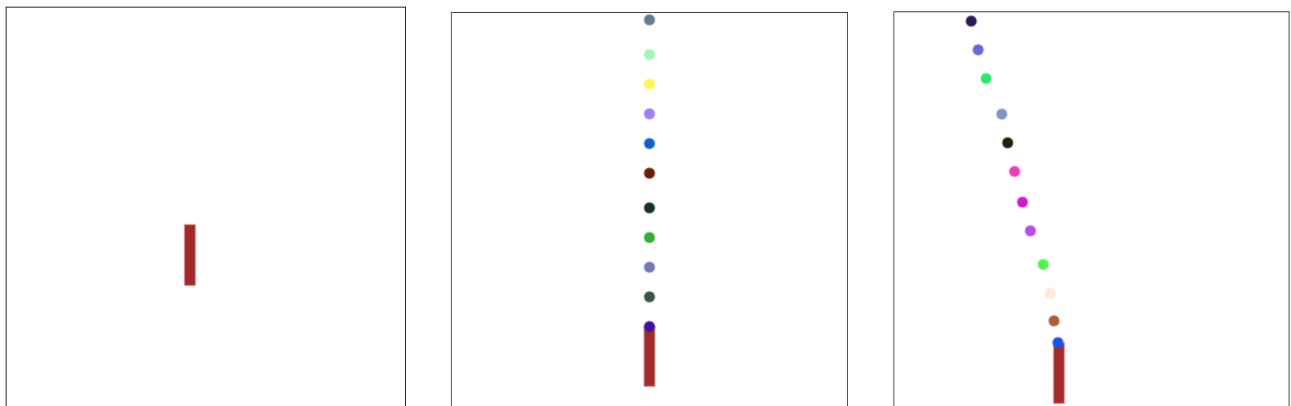
Implement a graphics program that introduces a magic wand to the canvas, designed as a rectangle with a **"brown"** fill color. It should have a height of 60 pixels, a width of 10 pixels, and starts with its top-left corner at the origin **(0,0)**. The wand should follow the mouse cursor in both directions so that its top-left corner attaches to the mouse's current coordinate with each **"mousemove"** event. Unlike what was required of you in **Breakout**, it's perfectly ok if the wand leaves the canvas when your mouse does.

When the user **clicks** anywhere on screen for the first time, sparks fly up from the wand and **float upwards** to the top of the window, moving 10 pixels every 10 milliseconds. The sparks are randomly colored circles with a 10-pixel diameter (notably identical to the wand's width), and one is introduced every 50 milliseconds. At the time of its creation, the spark will share the same top-left coordinate as the wand. When a spark moves off the top of the window—so that it's no longer visible—you should both remove the spark from the window, and terminate that spark's flying animation with **clearInterval**.

When the user **clicks** anywhere a second time, the wand **stops** emitting colorful sparks, but any existing sparks will continue flying up. A third click will cause sparks to be sprayed again, a fourth click stops the spraying, and so forth, so that a **"click"** event effectively **toggles the wand** between emitting and not emitting sparks. Along with event listeners to track the mouse (moves and clicks), your approach will require nested interval timers to support two animations: one to create each flying spark, and an inner one to keep moving that particular spark upward until it soars out of view.

You'll likely need some way of tracking whether sparks are currently flying or not, and a way to start and stop the animation of sparks being created, e.g., methods **setInterval** and **clearInterval**.

Though it's difficult to present animation on a physical page, screenshots of the magic wand in action are provided below ☺. On the left is the wand when no sparks are flying. In the middle is a stationary wand shooting varicolored sparks to the sky. On the right is a moving wand actively shooting sparks.



Present your implementation of **sparksFly**, and callback functions, on the next page. Along with timer functions, you may find the methods **randomColor**, **setLocation**, and **addEventListener** useful.

```
/* Constants (in pixels */
const GWINDOW_HEIGHT = 400;
const GWINDOW_WIDTH = 400;
const WAND_WIDTH = 10;
const WAND_HEIGHT = 60;
const SPARK_Y_VELOCITY = -10;
const SPARK_DIAMETER = 10;

/* Constants (in milliseconds) */
const FLYING_TIME_STEP = 10;
const CREATION_TIME_STEP = 50;

/* Program where you get to do your magic a la wizards of waverly place */
function sparksFly(){
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);

}
}
```

Extra space for Problem #2 if needed!

Problem 3: Shape-Shifting Cipher (Strings, 15 points)

Scytale is one of two things, depending on what you're more interested in, though both are menaces: (1) the scary shape-shifting villain of *Dune Part Three* (2026), played by Twilight heartthrob Robert Pattinson, or (2) an ancient Greek and Spartan cipher that relies on shape-shifting a text's letters around to scramble them.

Implement the function **scytale**, which pertains to the 2nd definition, of course ☺, and here's one way to interpret how it worked. For a given **plaintext**, and a given number of **columns**, we first **write the plaintext onto a grid with that many columns**, padding the text with **"X"**s as needed to fill up the last row.

As an example, say that in *The Odyssey*, King Odysseus has returned home to the Greek island of Ithaca. He may write the following message to his son Telemachus.

DADDYISBACKTELLYOURMOMMY (with spacing, "DADDY IS BACK TELL YOUR MOMMY")

Now, Odysseus has a lot of enemies, most of whom are suitors wanting to marry his wife Queen Penelope, and the main one is incidentally played by Robert Pattinson (Scytale himself 🙄). And so, he carefully disguise this message—24 letters long—in a grid with 5 columns, and thus 5 rows, writing letters from left to right, then top to bottom. This grid appears as below. **Note if the message is not a multiple of columns, we add "X" letters to it until it is.** Here, we add a single "X" to the end, to pad the plaintext to 25 letters.

D	A	D	D	Y
I	S	B	A	C
K	T	E	L	L
Y	O	U	R	M
O	M	M	Y	X

To produce the ciphertext, instead of reading the letters left-to-right, then top-to-bottom (which recovers the plaintext), **you invert the order: read top-to-bottom first, then left-to-right**. For the example above, the first column gives you **DIKYO**, the second gives **ASTOM**, and so on. Accordingly, the eventual ciphertext is:

DIKYOASTOMDBEUMDALRYCYCLMX

Had we used 6 columns, our grid would have 4 rows, with no need for padding as the message is 24 letters.

D	A	D	D	Y	I
S	B	A	C	K	T
E	L	L	Y	O	U
R	M	O	M	M	Y

And the resulting ciphertext would have been **DSERABLMDALODCYMYKOMITUY**.

Use the space on the next page to present your implementation of **scytale**, which given the **plaintext** and a positive number of **columns**, returns the ciphertext as achieved by this cipher. Happy encrypting!

*P.S. If you're curious how this worked, back in Ancient Greece, one would write the ciphertext on a narrow strip of parchment. Only by wrapping the strip in a spiral around a rod of specific diameter—so that exactly **columns** letters wrapped around—could one decrypt the message, by reading the letters aligned on each of the rod's sides, i.e. the rows of the grid. Without the correctly sized rod, the rows would form nonsense.*



```
const PAD_LETTER = "X";

function scytale(plaintext, columns) {
    // use the rest of this space for your implementation!
```

```
}
```

Extra space for Problem #3 if needed!

Problem 4: Paperless Rings (Arrays, 15 points)

"I like shiny things, but I'd marry you with paper rings" – Taylor Swift (song: Paper Rings, album: Lover)

Every autumn, thousands of students sign up for the **Marriage Pact**, a fun matchmaking activity to find a prospective partner among fellow participants by filling out a questionnaire. An algorithm then decides pairs based on proximity and compatibility to each other's answers, but sadly, inevitably, a few unlucky souls go unmatched ☹️. I was less fortunate than that: I once matched with my close friend's partner 😬.

You'll implement a basic matchmaker function **MarriagePact**. But instead of basing compatibility on elaborate, life-probing questions, **you'll simply look at Zodiac star signs to determine who can be matched** ☺️. Some of my friends do this in real life in fact. As such, the function's first two inputs are:

- **participants**: An array of two-element arrays of form **[name, sign]**, which are the name of a participant and their Zodiac star sign in that order. For instance,

```
participants = [{"Romeo", "LEO"}, {"Juliet", "LEO"}, {"Ennis", "VIRGO"},
                {"Peter", "LIBRA"}, {"MJ", "PISCES"}, {"Jack", "CAPRICORN"}]
```

- **signGroups**: An array of groups, where each is an array of signs that are mutually compatible with each other. Star signs from different groups are considered **not** compatible. As an example,

```
signGroups = [{"ARIES", "LEO", "SAGITTARIUS"}, // fire signs
               {"TAURUS", "VIRGO", "CAPRICORN"}, // earth signs
               {"GEMINI", "LIBRA", "AQUARIUS"}, // air signs
               {"CANCER", "SCORPIO", "PISCES"}] // water signs
```

The next two inputs are initially empty arrays, **matches** and **unmatched**, which we populate so that:

- **matches** becomes an **array of matched pairs** (each pair is a 2-element array of names). Each pair must be compatible, i.e. their star signs must be in the same group in **signGroups**.
- **unmatched** becomes an **array of names who ultimately went unmatched** ☹️. Sadness. Note that no two people in here can be compatible, otherwise they could have been matched!

Given the example **participants** and **signGroups** above, and **matches = []**, **unmatched = []**, a call to **MarriagePact(participants, signGroups, matches, unmatched)** can result in:

```
matches = [{"Romeo", "Juliet"}, {"Ennis", "Jack"}]
unmatched = ["Peter", "MJ"]
```

Note that Romeo and Juliet both have **"LEO"** signs, and that Ennis and Jack's signs, **"VIRGO"** and **"CAPRICORN"**, respectively, are in the same group in **signGroups**, making these couples compatible. Meanwhile, Peter and MJ's signs, **"LIBRA"** and **"PISCES"**, respectively, belong to different groups ☹️, so they couldn't be matched together. However, had the array of star sign groups instead been

```
signGroups = [{"LIBRA", "PISCES"}, // new group created for Peter + MJ <3
               {"ARIES", "LEO", "SAGITTARIUS"},
               {"TAURUS", "VIRGO", "CAPRICORN"},
               {"GEMINI", "AQUARIUS"},
               {"CANCER", "SCORPIO"}];
```

Then all six people can be paired up ☺, and everyone's happy now. The resulting arrays would be:

```
matches = [ ["Romeo", "Juliet"], ["Ennis", "Jack"], ["Peter", "MJ"] ]
unmatched = [ ]
```

Matches occur with this strategy: iterate over **participants** from index 0 up. For each one,



- If this participant has already been matched, skip the below, and go to the next iteration (*hint: you can use an array of Booleans to track who has and hasn't been matched yet*).
- Otherwise, look ahead in the **participants** array to see if anyone hasn't been taken yet and is sign compatible with this participant. If such a person is found, match them. That is, add the pair to the **matches** array and indicate internally they're both matched.
- Only if no match can be found for this participant, add them to the **unmatched** array.

The order of pairs in the **matches** array, order of names in each matched pair, and order of names in the **unmatched** array, does not matter. You may assume that the **participants** array is non-empty, that all names inside are distinct, and that everyone's star sign belongs to some group of **signGroups**.

Use the next page to write your implementation. You should modify the **matches** and **unmatched** arrays in place, as the function possesses no return value. As a hint, I suggest you write a helper function **isCompatible(zodiacOne, zodiacTwo, signGroups)**, which returns **true** only if the two Zodiac signs are in the same group, and thus marriage compatible, and returns **false** otherwise.

```
/* Creates pairs of star-crossed lovers / participants, and populates  
 * an array of matches ☺, and an array of unmatched participants ☹.  
 */  
function MarriagePact(participants, signGroups, matched, unmatched) {
```

```
}
```

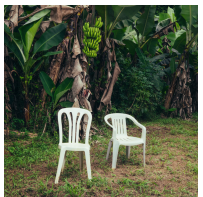
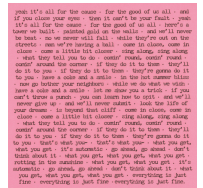
Extra space for Problem #4 if needed!

Problem 5: Spotify Wrapped (Data Structures, 15 points)

In case you're not a frequent Spotify user or music aficionado, *Spotify Wrapped* is an annual social media phenomenon where, around every December, **people share their top songs and musicians of the year, as summarized by their music listening platform**. For instance, in 2025, my most-listened-to song was Tate McRae's *"Sports Car"*, and my top artist was J-pop duo YOASOBI. No comment on my favorite album lol.

It's October right now, but you're impatient, and desperate to know who your top artist and song are. You scrape your streaming data manually off the web, which is organized into a JSON data structure like this.

```
let streamsByAlbum = {
  "you seem pretty sad for a girl so in love": {
    "artist": "Olivia Rodrigo",
    "year": 2026,
    "songs": [
      {"title": "expectations", "streams": 50},
      {"title": "stupid song", "streams": 30},
      {"title": "maggots for brains", "streams": 70},
    ]
  },
  "serena joy": {
    "artist": "Olivia Rodrigo",
    "year": 2026,
    "songs": [
      {"title": "serena joy", "streams": 50},
    ]
  },
  "DeBI TiRAR Más FOToS": {
    "artist": "Bad Bunny",
    "year": 2025,
    "songs": [
      {"title": "EoO", "streams": 80},
      {"title": "DtMF", "streams": 20},
      {"title": "NUEVAYoL", "streams": 20},
    ]
  },
  "Nevermind": {
    "artist": "Nirvana",
    "year": 1991,
    "songs": [
      {"title": "Smells Like Teen Spirit", "streams": 100},
      {"title": "Breed", "streams": 10},
    ]
  }
}
```



Each key of **streamsByAlbum** is the title of an album, and the corresponding value is a nested object with three keys: **artist**, the name of who created the album, e.g., **year**, the year the album was released, and **songs**, which maps to a list that describes each song the user listened to from that album. Specifically, each element on this list is an object with two keys: **title**, which maps to the song's title, and **streams**, which maps to a positive integer for how many times the user streamed / listened to that song this year.

In the example above, **Nirvana's galactic hit "Smells Like Teen Spirit"** (described as the anthem of Generation X, hi Jerry!!) **was our most-listened song with 100 streams**. And tallying up the total number of streams per artist, we have **"Olivia Rodrigo"** with 200 streams, **"Bad Bunny"** with 120 streams, and **"Nirvana"** with 110 streams—so **Olivia Rodrigo is the winner and by far our most-listened-to artist** 😊.

In this (teen) spirit, implement the function **MusicWrapped**, which accepts as input a data structure **streamsByAlbum** like the one above, and returns an object with two keys, **topArtist** and **topSong**. These map to our most streamed artist and song, respectively. For the example above, we'd return

```
{  
  "topArtist": "Olivia Rodrigo",  
  "topSong": "Smells Like Teen Spirit"  
}
```



You may assume that **streamsByAlbum** contains at least one album, and at each album contains at least one song. If there are two or more artists tied for most streamed, you can return any one of them, and the same applies for songs. And as an implementation hint, **to identify the top artist, you'll almost certainly want to build up a JavaScript map of each artist to the total number of streams across all their songs**.

Present your implementation of **MusicWrapped** on the next page.

```
/* Given a user's streaming data, returns an object with that user's
 * most-streamed song, and most-streamed artist across all their songs.
 */
function MusicWrapped(streamsByAlbum) {
```

```
}
```