

CS 106AX

Programming Methodologies in JavaScript and Python

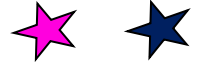
Autumn 2026



Ben's Extra Midterm Practice



Written by Ben Yan.



Here's an additional set of 106AX practice problems that I wrote or curated, as a vivarium for the even more colorful and chaotic ideas circling in my brain this time around. While they didn't quite make it into the starting lineup that is **Ben's Practice Midterm Exam**, I still think they're of reasonable quality ☺.

Problems

1. Never-Ending Stories
2. Blackpink in Your Area
3. Rail Fence Cipher
4. What the F—
5. U.S. National Debt
6. Nomad Palindrome
7. Hung Parliament
8. Great CS106AX Bake Off!

Topics

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

In case you didn't get the memo from the table of contents above, midterm is all JavaScript, no Python! Also, you're more than welcome to peruse the Interactive Grading (IG) and section problems for practice.

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.

Problem 1: Never-Ending Stories (JavaScript Expressions and Tracing)

1. Compute the value of each of the following JavaScript expressions.

`(20 + 7) % 24 + 24 % 12`

`1 + 2 + 3 + 4 === 10 || 1234 / 0 === 1234`

`10 + 6 + "YO" + 3 * 4 + 9 + "ASOBI"`

2. Assume the function **chaseYourDreams** is defined as presented below. The code is, of course, narratively and poetically nonsense, and you just want to do a faithful JavaScript trace.

```
function chaseYourDreams(dreamOne, dreamTwo){
  let resilience = 0;
  let failure = 1;
  while (dreamOne > 0) {
    let learning = dreamOne % 10 + dreamTwo % 10;
    for (let i = 0; i < failure; i++) {
      resilience += learning;
    }
    failure *= 10;

    let belief = Math.floor(dreamOne / 10);
    dreamOne = Math.floor(dreamTwo / 10);
    dreamTwo = belief;
  }
  return resilience;
}
```

What's the value returned by a call to **chaseYourDreams (2048, 4096)** ?

3. Assume the following (silly) definitions of the functions **sillyString** and **sillyBanz**.

```
function sillyString(string) {
  let spray = string.charAt(0);
  let rainbow = string.substr(1, string.length - 1);
  let silly = sillyBanz(spray, rainbow);
  return silly + spray.toUpperCase();
}
```



```
function sillyBanz(friend, bracelet) {
  let memories = "abc".charCodeAt(1) - friend.charCodeAt(0);
  let ruined = sillyMe(memories);
  function sillyMe(why){
    let again = bracelet.indexOf(friend);
    return memories + again + why;
  }
  return bracelet.substr(0, ruined) + ruined;
}
```

What output is printed by a function call to **sillyString("agreatpaint")**?

Problem 2: Blackpink In Your Area (Graphics and Animation)

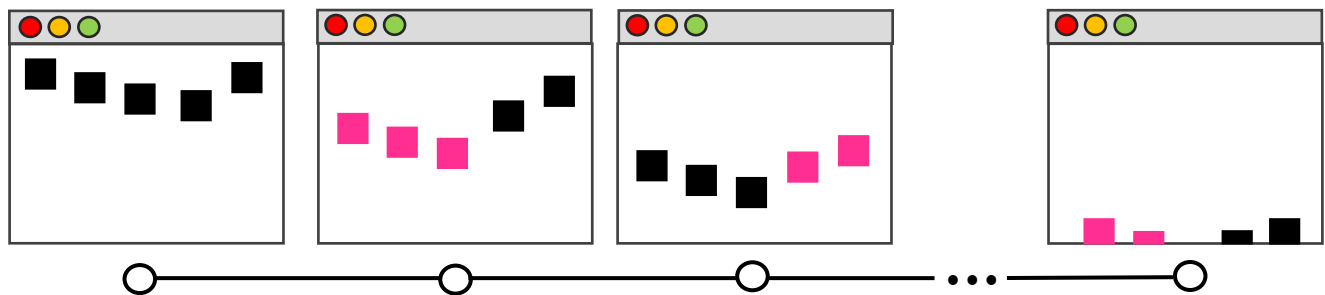
"How You Like That?" – one of Blackpink's signature songs. Also Jerry when you're done with his exam 🐱.

Implement a graphical program that introduces a square whenever and wherever the user clicks on the graphics window. Each square has a width and height of **50** pixels, and should be centered at the mouse click. The square color should start off as **"Black"**, though it won't stay that color for the entire runtime. Let me explain 😊.



Once introduced, each square should begin falling toward and eventually beyond the bottom of the graphics window. When the square is no longer visible on-screen, it should be removed from the window. As the square falls, every **1000** milliseconds (1 second), its color changes to **"Pink"** if its currently **"Black"**, and to **"Black"** if currently **"Pink"** 😊.

Pink squares fall faster than black squares, at a rate of **2** pixels every **50** milliseconds, whereas black squares fall at a rate of **1** pixel every **50** milliseconds. Neither black nor pink squares move horizontally, only vertically downward. While it's hard to capture animation on a static page, here's a simulated chronological timeline of frames below, from the earliest frame on the left, to the most recent frame on the right.



For context, I clicked at five points near the top of the window, going left to right, and just let the squares fall—and like the autumn leaves, they change color 😊. Though I didn't click after that, the program should be able to handle and respond to more mouse clicks, even when the previous squares are falling.

A few more things worth noting:

- You're aware of the **setColor(*color*)** method, which sets the border color—and if otherwise unspecified, also the fill color—of a graphical object. There exists also the **getColor** method, which returns the receiving object's color as a string, e.g., **"Black"**, **"Pink"**, etc.
- You've seen and leveraged the **setInterval(*func*, *timestep*)** method for timer animations, which schedules the **func** function to execute every **timestep** milliseconds until a corresponding **clearInterval** is called. Ensure that when a square is removed from the window (after falling through the bottom), any timer animation(s) you've created for that square are cleared.

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

```
/* Constants (in pixels) */
let GWINDOW_WIDTH = 500;
let GWINDOW_HEIGHT = 500;
let SQUARE_DIMENSION = 30;
let BLACK_SQUARE_VY = 1;
let PINK_SQUARE_VY = 2;

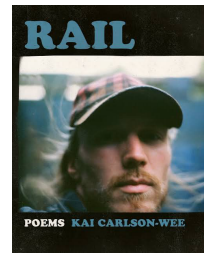
/* Constants (in milliseconds) */
let FALLING_TIME_STEP = 50;
let COLOR_CHANGE_TIME_STEP = 1000;

function BlackpinkInYourArea() {
    let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
    // Present the rest of this graphical program's implementation here!

}
```

Problem 3: Rail Fence Cipher (Strings)

A rail fence cipher encrypts a plaintext message by writing the characters up and down diagonally on successive “rails”. We start writing downwards and to the right, move upwards when we’ve hit the bottom rail, then move downwards when we hit the top rail, and so on in a zig-zag pattern chugging relentlessly, like a train, from left to right.



For instance, if our message is "HELLOWORLD", and the number of rails is 3, we can draw:

```

H   _   _   _   O   _   _   _   L   _   _   _
_   E   _   _   L   _   _   _   W   _   _   _
_   _   _   L   _   _   _   O   _   _   _
  
```

The original message can be reading the letters left to right across rails. However, **to produce the encrypted message, we instead read first along rows**, left-to-right, then down along successive rails. As such, the encrypted message here would be "HOLELWRDLO", which yes, isn't exactly good, effective security ☺.

But with a longer message, "NOLUKEIAMYOURFATHER"—yes, I know that's not the quote—and more rails,

```

N   _   _   _   M   _   _   _   H   _   _   _
_   O   _   _   _   A   _   _   _   Y   _   _   _   T   _   _   _   E
_   _   _   L   _   _   _   I   _   _   _   O   _   _   _   A   _   _   _   R
_   _   _   U   _   _   _   E   _   _   _   U   _   _   _   F
_   _   _   K   _   _   _   _   _   _   R
  
```

The encrypted message using these 5 rails would be the more well disguised "NMHOAYTELIOARUEUFKR".

Your task is to implement `railFenceEncrypt`, which accepts an input **message** string and the **number of rails**, and returns a string representing the encryption of **message** using the rail fence cipher. While the programming logic is easier with the use of ancillary data structures such as JavaScript arrays or maps, as a bonus challenge, try only using strings and integer types in your code ☺. I promise that it's entirely possible!

Use the space on the next page to skateboard your pencil around and thereby present your implementation.



```
function railFenceEncrypt(message, numRails) {  
    // Present your code to encrypt the message here!
```

```
}
```

Problem 4: What the F— (Strings)

Swearing and profanity can be a touchy topic for people, especially to Captain America (*language!*) and many parents of young children. You're tasked with writing a function **bleep** that, as its name suggests, "bleeps" out bad words ☺ or extremely colorful language. This function takes as input two parameters:

- **text**: A string that can have zero, one, or arbitrarily many bad words inside.
- **replacement**: A (probably) more innocent string that you replace each bad word with, e.g., the TV sound "BLEEP", "XXXX", "*****", the growling grawlix "&S!#%", you name it.

and returns a new string where **each bad word in text is substituted by the replacement substring**. Bad words are identified in the string by double square brackets, "[[", "]]", on each side, such as

```
[[bad word]], [[heck]], [[job]], [[imbecile]], [[grass mud horse]]
```

For instance, given the inputs:

```
text = "what the [[heck]] Ben? what in the [[flying fck]] are you doing"
replacement = "BLEEP"
```

A call to **bleep(text, replacement)** should return the new, somewhat kinder string ☺.

```
"what the BLEEP Ben? what in the BLEEP are you doing"
```

Note the brackets are gone. Some more examples, for your entertainment, are jovially provided below.

JavaScript Console

```
> bleep("The name's [[bond]], James [[bond]].", "BLEEP")
The name's BLEEP, James BLEEP.

> bleep("They call him [[Lord Voldemort]]", "He who shall not be named")
They call him He who shall not be named

> bleep("apply to [[j*bs]], [[empl*yment]], and [[w*rk]]", "coterm")
apply to coterm, coterm, and coterm

> bleep("[[Jerry Cain]]@stanford.edu", "poohbear")
poohbear@stanford.edu

> bleep("I do not need to swear to express my feelings ☺", "XXX")
I do not need to swear to express my feelings ☺
```



Present your implementation on the next page. You may assume that the **replacement** string itself contains no bracket characters, and that each "[[" in **text** is paired with its own subsequent "]]".

Hints: You'll likely want a **while True** loop to run until you can't find a "[[" substring anymore. Also, you can use the method **str.indexOf(substr)** to find the index where a substring **substr** first occurs in string **str**. If the substring is not found, then **indexOf** will return a sentinel value of **-1**.

```
/* Given a text string possibly full of bad, colorful words, creates a new
 * string where all bad words are substituted for a mundane replacement 🙄.
 */
function bleep(text, replacement) {
    // Present your implementation, and try not to swear in the process!
```

```
}
```

Problem 5: U.S. National Debt (Arrays)

Based on a practice problem written in Python from CS106A many years ago (like when I was a frosh)

With the US National Debt now well over 37 trillion dollars, you decide to implement **Big Numbers** in JavaScript, which enable you to represent an arbitrarily large positive integer value using an array of digits.

For instance, we could represent the integer **37,918,570,029,078** (the astonishing national debt of nearly 38 trillion dollars, which I can verify seeing on U.S. debt clock website last night) as the array

[3, 7, 9, 1, 8, 5, 7, 0, 0, 2, 9, 0, 7, 8]

However, to allow you to more easily add such numbers, **you store the digits of the Big Number in reverse**, such that the ones digit occupies index **0**, the tens digit occupies index **1**, the hundreds digit occupies index **2**, and so forth. As such, the debt number above would actually be represented as the array:

element	8	7	0	9	2	0	0	7	5	8	1	9	7	3
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Note the array should have exactly as many elements as the number of digits in the number it represents.

Suppose now we have a **bigNumOne** representing **9564**, **bigNumTwo** representing **867**, and we want to compute the sum of both numbers (metaphorically, if we're piling more owed money on the national debt).

bigNumOne

element	4	6	5	9
index	0	1	2	3

bigNumTwo

element	7	6	8
index	0	1	2



We can add the ones digits in the **0th** index, then the tens digits in the **1st** index, and so forth, to produce the resulting sum **10431**, as shown in the five-digit **Big Number** array below.

sum	element	1	3	4	0	1
	index	0	1	2	3	4

Your job is to implement a function **addBigNumbers**, which accepts two **Big Numbers** (arrays of digits listed in backward order), and returns a new **Big Number** array that represents the sum of the two numbers passed in. **You should use the algorithm described above to add the numbers digit by digit, propagating any necessary carries to the next digit.** Your function should not modify the arrays passed in.

```
/* Accepts two big number arrays (digits in reverse order) as input, and
 * returns a big number array that represents the sum of the input numbers.
 *
 * >> addBigNumbers([3, 2, 1], [4, 3, 2]) => [7, 5, 3] // 123 + 234 = 357
 *
 * >> addBigNumbers([1, 9, 9], [0, 1]) => [1, 0, 0, 1] // 991 + 10 = 1001
 */
function addBigNumbers(bigNumOne, bigNumTwo){
    // Use the space here to present your implementation!
```

```
}
```

Problem 6: Nomad Palindrome (Arrays)

Inspired by Kai Carlson-Wee's poem, "Nomad Palindrome", a beautiful work that reads palindromically ☺.

In mathematics, a palindrome is an integer whose digits read the same from left to right as right to left. For instance, **12321**, **757**, **99**, and all single-digit numbers are palindromes, whereas **1234** and **67** are not.

Your task is to implement the function **generate**, which accepts as input an integer **maxDigits**, and generates all palindrome numbers with at most **maxDigits** number of digits, and each digit between **1–9**. To clarify, for ease of programming, **we'll exclude numbers with any zero digits**, e.g., **20**, **101**, **9000**.

However, instead of mixing all these palindromes in a single array, **return an array of `maxDigits + 1` lists**, where **each list or inner array contains all palindromes, ordered in increasing numeric order, with the same number of digits**. That is, the 0th list is empty, as there are no integers—let alone palindromes—with zero digits, the 1st list is **[1, 2, 3, 4, 5, 6, 7, 8, 9]** (since all single-digit numbers are palindromes), and the 2nd list is **[11, 22, 33, 44, 55, 66, 77, 88, 99]**. The 3rd list is all three-digit palindromes, and so forth, up to the final list, which containerizes all palindromes with precisely **maxDigits** digits, e.g., if **maxDigits = 5**, this list would be **[11111, 11211, 11311, ...]**.

Here's an iterative algorithm you can deploy to generate each list, one at a time. We rely on the elegant truth that, for a **k**-digit number to be a palindrome, only these two properties need to be true.

- The leftmost and rightmost digits are the same.
- The middle **(k - 2)** digits, treated as its own number, is a palindrome.

Take for instance, the 5-digit number **12321**. The left and right digits are both **1**, and the middle 3-digit number, **232**, is a palindrome, so **12321** is a palindrome. As such, we can generate **k**-digit palindromes from an existing pool of **(k-2)** digit palindromes, by gluing a new, identical digit to the left and right sides.

To demonstrate, for example, how you can generate 4-digit palindromes from the list of 2-digit palindromes,

- By adding a pair of **1** digits, from **11, 22, 33, ...**, we can generate **1111, 1221, 1331**, and so on.
- By adding a pair of **2** digits, from **11, 22, 33, ...**, we can generate **2112, 2222, 2332**, and so on.
- By adding a pair of **3** digits, from **11, 22, 33, ...**, we can generate **3113, 3223, 3333**, and so on.

And we can proceed similarly if our new pair of digits—added to the left and right sides—is a **4** or **5** or **6** and so on up to **9**. Note you can use **Math.pow(num, exp)** to compute exponents when needed, and you'll want to make sure any loops you write are organized so that each resulting list is in increasing order.

You are guaranteed the input **maxDigits** is at least 2, and the provided skeleton code already adds the 0th, 1st, and 2nd palindrome lists to the outer array. Your goal is to generate all lists beyond that as necessary ☺.

Use the vast, nomadic landscape of the next page to present your implementation. (*P.S. I promise it's not that difficult to include palindromic numbers with some zero digits, e.g., 10101, just a little clunky for an exam setting. I'll leave it as a bonus challenge if you'd like to figure out how to do this, I believe in you!!*)

```
function generate(maxDigits) {  
  let palindromes = []  
  palindromes.push([]); // no zero-digit numbers or palindromes  
  palindromes.push([1, 2, 3, 4, 5, 6, 7, 8, 9]);  
  palindromes.push([11, 22, 33, 44, 55, 66, 77, 88, 99]);  
  
  // present your implementation in the space given below!
```

```
}
```

Here's the poem I was inspired by, in case you're looking for inspiration of your own to code this up ☺.

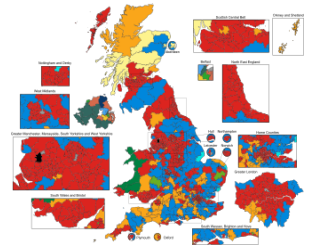
Nomad Palindrome

Maps, dogs, rail, stars, a Warsaw door,
no sleep, no moods,
I lived on flow, on speed, on spit-faced lonely tons,
red rum, Tums, on bad dessert, sands,
eye-level, dim as time,
stab on, sway on, span on, Erewhon,
drawn onward,
nowhere, no naps, no yaws, no bats
(emits a mid-level E), yes, DNA stress, ED, dab,
no smut, murders, no Tylenol, decaf, tips,
no deeps, no wolf, no devil-is-doom,
on peels, on rood,
was raw as rats, liars, God, Spam.

*A Pushcard Prize winning poem by Kai Carlson-Wee,
published in the AGNI literary magazine*

Problem 7: Hung Parliament (Working with Data Structures)

Every 5 years or so, the United Kingdom holds a general election, when millions of voters choose who'll represent their constituency, e.g., *Edinburgh North*, in Parliament. The latest election in 2024 was a thundering victory for the Labour Party, who won a landslide majority (411 of 650 total seats), and a shambolic defeat for the Conservative Party. The 2010 election was a completely different story. **No party achieved a majority of seats** (the Conservatives had 306, and Labour had 258), **resulting in a so-called *hung parliament***, which can get spicy.



Imagine that you're studying overseas at the Stanford in Oxford program (*which I highly endorse, even if there was much drama during my term*), and election night is happening. You've been asked to analyze results, and are given a large JSON object with each constituency's votes, which may appear like this.

```
let electionData = {
  "Aberafan Maesteg": [
    {"candidate": "Colin Deere", "party": "Plaid Cymru", "votes": 4719},
    {"candidate": "Stephen Kinnock", "party": "Labour", "votes": 17838},
    {"candidate": "Mark Griffiths", "party": "Reform", "votes": 7484},
  ],
  "Aberdeen North": [
    {"candidate": "Kristy Blackman", "party": "SNP", "votes": 14533},
    {"candidate": "Lynn Thompson", "party": "Labour", "votes": 12773},
    {"candidate": "Gillian Tebberen", "party": "Conservative", "votes": 5881},
  ],
  ... several constituencies omitted for conciseness
  "Holborn and St. Pancras": [
    {"candidate": "Andrew Feinstein", "party": "Independent", "votes": 7312},
    {"candidate": "Keir Starmer", "party": "Labour", "votes": 18884},
  ],
  ... several constituencies omitted for conciseness
  "York Outer": [
    {"candidate": "Julian Sturdy", "party": "Conservative", "votes": 13770},
    {"candidate": "John Crispin-Bailey", "party": "Reform", "votes": 5912},
    {"candidate": "Luke Charters", "party": "Labour", "votes": 23161},
    {"candidate": "Andrew Hollyer", "party": "LibDem", "votes": 5496},
  ],
};
```

The keys of `electionData` are the names of U.K. constituencies, e.g., "**Aberafan Maesteg**", "**Aberdeen North**", "**York Outer**", and each corresponding value is a list of objects. Each inner object has precisely 3 keys: "**candidate**", the name of a candidate in that constituency, "**party**", the political party they ran under, and "**votes**", an integer count of how many votes they received.

The U.K. has a *first-past-the-post* system where, **for each constituency, the candidate that wins the greatest number of cast votes—even if it's not a majority—wins that constituency's seat**. For instance, "**Kristy Blackman**" of the "**SNP**" (Scottish National Party) won the seat for "**Aberdeen North**", as her **14533** votes (~35%) surpassed any other candidate in that constituency. You can leverage this to identify which party won each constituency (by looping over the list of data objects, and keeping track of some best-so-far candidate and vote count), and thus how many seats each political party controls in Parliament. One useful way to capture this information is by building a **party-to-seats map**, e.g.,

```
{"Labour": 412, "Conservative": 121, "LibDem": 72, "SNP": 9, "SF": 7, ...}
```

Your task is, ultimately, to implement the function `isHungParliament`, which accepts as an input a large JSON object `electionData` structured like the one above. This function returns `true` only if the election resulted in a **hung parliament (that is, no party achieved a majority of seats in Parliament)** and returns `false` otherwise. Instead of assuming a fixed 650 seats in total, you should check if any party in Parliament has strictly greater than $0.5 * \text{electionData.length}$ number of seats.

You may assume that, **for each constituency**, there are no ties for first place, and that at least one, and possibly many, candidates are running, but no two are from the same party. I recommend that you build the **party-to-seats** map from `electionData`, and as a small example, should the seats distribution be:

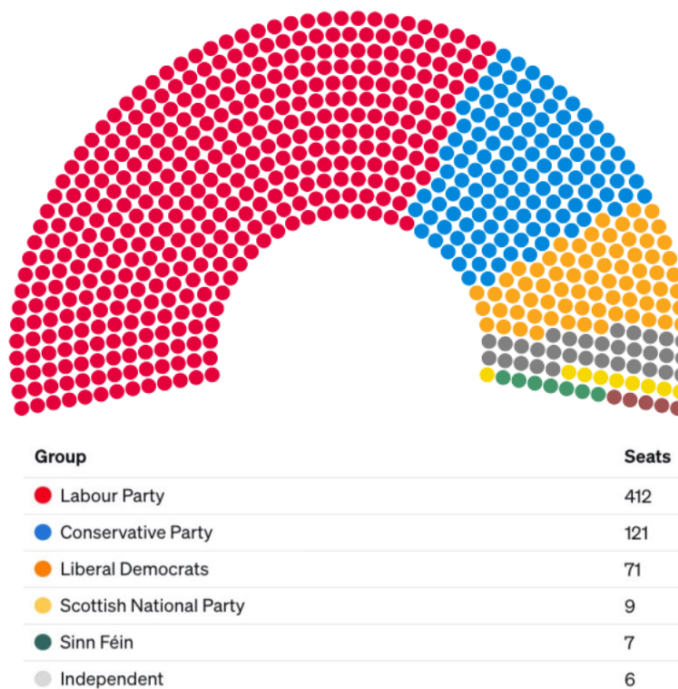
```
partyToSeats = {"Labour": 7, "Conservative": "2", "SNP": "1"}
```

then **"Labour"** would have a majority of the 10 total seats, so **no** hung parliament. Conversely, if

```
partyToSeats = {"Labour": 5, "Conservative": "3", "SNP": "2"}
```

then no party would have a majority (even though Labour is 1 away), so we have a **hung parliament**.

Use the space on the following page to present your implementation of `isHungParliament`.



```
function isHungParliament(electionData) {  
    // Present your implementation here!
```

```
}
```

Problem 8: Great CS106AX Bake Off (Working with Data Structures)

Note: This is, as of right now, fiction, but if you'd like to organize a class-wide baking event, let me know!

We as a class are holding a great 106AX bake off contest, in the spirit of the popular British TV show. To prepare, you run to the Trader Joes grocery store in Palo Alto, collect several ingredients, and place them in a pantry, as represented by the JavaScript object / map below. The names or keys are ingredients, and the values are amounts—metric system, of course.



```
let pantry = {
  "flour": 400,
  "sugar": 200,
  "salt": 10,
  "chocolate": 100,
  "vanilla": 50,
};
```

To prepare for this grand bake off, you will make several cake recipes / orders, and each cake recipe is placed in an array of objects as displayed below. Each object inside, like the pantry object above, has keys that are ingredient names, and the corresponding values are the amounts needed to bake the cake.

```
let cakeOrders = [
  {"flour": 40, "sugar": 20, "vanilla": 4},
  {"flour": 20, "salt": 10, "chocolate": 5, "vanilla": 5},
  {"flour": 60, "sugar": 40, "chocolate": 10},
  ... more cake orders formatted like the ones above
];
```

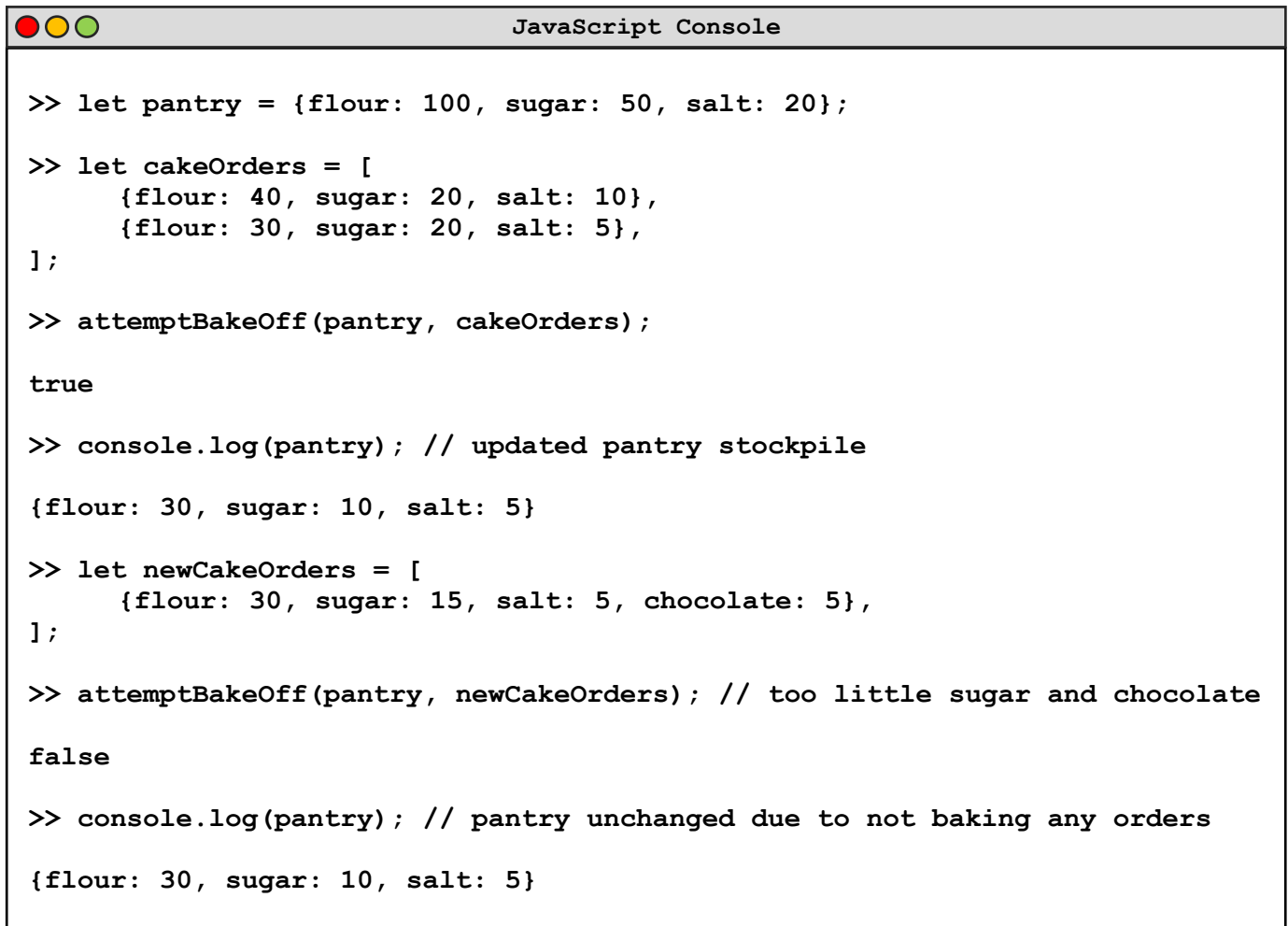


Your task is to implement the function **attemptBakeOff**, which accepts two parameters as input: a **pantry** map formatted like the one above, and an array **cakeOrders** of one or more recipes. This function should determine if the pantry has sufficient ingredient amounts to **make all the orders**, and:

- If the pantry **can** make the orders, **attemptBakeOff** returns **true**, and before returning, **modifies the passed-in pantry** to decrement the ingredient amounts as required by the cake orders, as though we're gone ahead and baked all of them ☺.
- If the pantry **cannot** make the orders (i.e. the total amount of at least one ingredient across all orders exceeds the pantry's supply of that ingredient), then **attemptBakeOff** returns **false**, and **does not modify** the passed-in pantry. We don't even bake a single cake if we can't bake all of them ☹.

You can assume that if an ingredient name is not stored in the pantry object, its amount is zero. As an **implementation hint**, as you iterate over **cakeOrders**, you should build up a map of ingredients to the cumulative amount of that ingredient required by the cake orders. You should wait to decrement the ingredient amounts in pantry unless you've confirmed the pantry has sufficient amounts of each ingredient.

To aid your creative imagination, an example run in the JavaScript console is shown on the next page.

A screenshot of a web browser's JavaScript Console. The window has a title bar with three colored circles (red, yellow, green) on the left and the text "JavaScript Console" in the center. The main area of the window contains a series of JavaScript commands and their outputs, all in a monospaced font. The commands include variable declarations for 'pantry' and 'cakeOrders', function calls to 'attemptBakeOff', and 'console.log' statements. The outputs are 'true' and 'false' for the function calls, and object literals for the logged 'pantry' variable.

```
>> let pantry = {flour: 100, sugar: 50, salt: 20};

>> let cakeOrders = [
    {flour: 40, sugar: 20, salt: 10},
    {flour: 30, sugar: 20, salt: 5},
];

>> attemptBakeOff(pantry, cakeOrders);

true

>> console.log(pantry); // updated pantry stockpile
{flour: 30, sugar: 10, salt: 5}

>> let newCakeOrders = [
    {flour: 30, sugar: 15, salt: 5, chocolate: 5},
];

>> attemptBakeOff(pantry, newCakeOrders); // too little sugar and chocolate

false

>> console.log(pantry); // pantry unchanged due to not baking any orders
{flour: 30, sugar: 10, salt: 5}
```

Use the space on the next page to present your implementation of **attemptBakeOff** 😊.

```
function attemptBakeOff(pantry, cakeOrders) {  
  // Present your implementation here!
```

```
}
```