**The ways [tech interviews are being carried out](https://dev.to/vbordo/if-you-could-change-one-thing-about-the-technical-interview-process-what-would-it-be-3lli) has been at the center of much controversy for a while now. It’s a sensitive topic, especially when it comes to coding challenges.**

Not all companies use the same screening process, but for the most part, expect to be asked to solve a coding challenge, either on a suitable platform or on the dreaded whiteboard.

One complaint that’s usually made against coding challenges is that they’re mostly irrelevant to the day-to-day tasks the actual job requires. Especially when it comes to front-end interview questions, sometimes it’s curious how what’s missing in the interview are just front-end-related questions on things like browser compatibility, layout methods and DOM events. While this can be true, those who favor this approach and are responsible for hiring in these companies often say something like this:

I’d rather hire a smart person and teach them X than hire someone who knows everything about X but lacks creativity, logic, and reasoning. — [Interviewing as a Front-End Engineer in San Francisco](https://css-tricks.com/interviewing-front-end-engineer-san-francisco/)

Whatever we feel about the way candidates are screened for dev jobs, at the time of writing, coding challenges are still a big part of the interview process.

In this article, I’m going to show how you can tackle five common coding challenges you might be asked when interviewing for a JavaScript or front-end Junior Developer position. They’re not among the hardest ones you could come across in the interview process, but the way you approach each of them could make the difference between success and failure.

**Pointers on Tackling Coding Challenges for Your Tech Interview**

Before diving into the challenges, let’s go through some tips about how you could approach your tech interview.

* **Put in the time to prepare**. Make your priority to research, learn less familiar topics, and practice a lot. If you haven’t got a Computer Science background, make sure you get familiar with some fundamental topics related to **algorithms and data structures**. There are online platforms, both free and paid, that offer great ways to practice your interview skills. [GeeksforGeeks](https://www.geeksforgeeks.org/), [Pramp](https://www.pramp.com/" \l "/), [Interviewing.io](https://interviewing.io/), and [CodeSignal](https://codesignal.com/) are just a few of my favorite resources.
* **Practice thinking aloud** when you’re trying to come up with a solution. In fact, talking through your thought process in an interview setting is preferable to spending all the available time writing down your solution in total silence. Your words will give the interviewer a chance to help you if you’re about to take a wrong turn. It also highlights your communication skills.
* **Understand the problem before starting to code**. This is important. Otherwise, you might be wasting time thinking about the wrong problem. Also, it forces you to think about questions you may ask your interviewer, like edge cases, the data type of inputs/outputs, etc.
* **Practice writing code by hand**. This helps you get familiar with the whiteboard scenario. A whiteboard doesn’t provide the kind of help that your code editor provides — such as shortcuts, autocomplete, formatting, and so on. When preparing, try writing down your code on a piece of paper or on a whiteboard instead of thinking it all up in your head.

**Common Coding JavaScript Challenges**

It’s likely that you’ve come across one or more of the challenges I’ve listed below, either during a job interview or while practicing your JavaScript skills. What better reason is there for getting really good at solving them?

Let’s get cracking!

**#1 Palindrome**

A palindrome is a word, sentence or other type of character sequence which reads the same backward as forward. For example, “racecar” and “Anna” are palindromes. “Table” and “John” aren’t palindromes, because they don’t read the same from left to right and from right to left.

**Understanding the challenge**

The problem can be stated along the following lines: given a string, return true if the string is a palindrome and false if it isn’t. Include spaces and punctuation in deciding if the string is a palindrome. For example:

palindrome('racecar') === true

palindrome('table') === false

**Reasoning about the challenge**

This challenge revolves around the idea of reversing a string. If the reversed string is the same as the original input string, then you have a palindrome and your function should return true. Conversely, if the reversed string isn’t the same as the original input string, the latter is not a palindrome and your function is expected to return false.

**Solution**

Here’s one way you can solve the palindrome challenge:

const palindrome = str => {

// turn the string to lowercase

str = str.toLowerCase()

// reverse input string and return the result of the

// comparisong

return str === str.split('').reverse().join('')

}

Start by turning your input string into lower case letters. Since you know you’re going to compare each character in this string to each corresponding character in the reversed string, having all the characters either in lower or upper case will ensure the comparison will leave out this aspect of the characters and just focus on the characters themselves.

[Report Advertisement](https://www.sitepoint.com/5-common-coding-interview-challenges/#report-ad)

Next, reverse the input string. You can do so by turning the string into an array using the String’s [.split()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split) method, then applying the Array’s [.reverse()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse) method and finally turning the reversed array back into a string with the Array’s [.join()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join) method. I’ve chained all these methods above so the code looks cleaner.

Finally, compare the reversed string with the original input and return the result — which will be true or false according to whether the two are exactly the same or not.

**#2 FizzBuzz**

This is a super popular coding challenge — the one question I couldn’t possibly leave out. Here’s how you can state the problem.

**Understanding the challenge**

The FizzBuzz challenge goes something like this. Write a function that does the following:

* console logs the numbers from 1 to n, where n is the integer the function takes as its parameter
* logs *fizz* instead of the number for multiples of 3
* logs *buzz* instead of the number for multiples of 5
* logs *fizzbuzz* for numbers that are multiples of both 3 and 5

Example:

fizzBuzz(5)

Result:

// 1

// 2

// fizz

// 4

// buzz

**Reasoning about the challenge**

One important point about FizzBuzz relates to how you can find multiples of a number in JavaScript. You do this using the [modulo or remainder operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Remainder_%28%29), which looks like this: %. This operator returns the remainder after a division between two numbers. A remainder of 0 indicates that the first number is a multiple of the second number:

12 % 5 // 2 -> 12 is not a multiple of 5

12 % 3 // 0 -> 12 is multiple of 3

If you divide 12 by 5, the result is 2 with a remainder of 2. If you divide 12 by 3, the result is 4 with a remainder of 0. In the first example, 12 is not a multiple of 5, while in the second example, 12 is a multiple of 3.

With this information, cracking FizzBuzz is a matter of using the appropriate conditional logic that will lead to printing the expected output.

**Solution**

Here’s one solution you can try out for the FizzBuzz challenge:

const fizzBuzz = num => {

for(let i = 1; i <= num; i++) {

// check if the number is a multiple of 3 and 5

if(i % 3 === 0 && i % 5 === 0) {

console.log('fizzbuzz')

} // check if the number is a multiple of 3

else if(i % 3 === 0) {

console.log('fizz')

} // check if the number is a multiple of 5

else if(i % 5 === 0) {

console.log('buzz')

} else {

console.log(i)

}

}

}

The function above simply makes the required tests using conditional statements and logs out the expected output. What you need to pay attention to in this challenge is the order of the if … else statements. Start with the double condition first (&&) and end with the case where no multiples are found. This way, you’ll be able to cover all bases.

**#3 Anagram**

A word is an anagram of another word if both use the same letters in the same quantity, but arranged differently.

**Understanding the challenge**

You can state this challenge in the following terms: write a function that checks if two provided strings are anagrams of each other; letter casing shouldn’t matter. Also, consider only characters, not spaces or punctuation. For example:

anagram('finder', 'Friend') --> true

anagram('hello', 'bye') --> false

**Reasoning about the challenge**

The first thing to notice is that you’ll need to compare each letter in the two input strings, not only in terms of which letters they are but also how many of each letter there are in both strings. A way of mapping this information would look something like this:

finder --> f: 1 friend --> f: 1

i: 1 r: 1

n: 1 i: 1

d: 1 e: 1

e: 1 n: 1

r: 1 d: 1

A suitable data structure to hold the anagram data would be a JavaScript object literal: the key is the letter character and the value is the number of times that letter is contained in the given string.

Next, pay attention to these edge cases:

* make sure letter casing isn’t factored into the comparison; just transform both strings into either lower case or upper case
* leave everything that isn’t a character out of the comparison; a good option would be to use a [regular expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp).

**Solution**

Here’s how you could approach the Anagram challenge:

// helper function that builds the

// object to store the data

const buildCharObject = str => {

const charObj = {}

for(let char of str.replace(/[^\w]/g).toLowerCase()) {

// if the object has already a key value pair

// equal to the value being looped over,

// increase the value by 1, otherwise add

// the letter being looped over as key and 1 as its value

charObj[char] = charObj[char] + 1 || 1

}

return charObj

}

// main function

const anagram = (strA, strB) => {

// build the object that holds strA data

const aCharObject = buildCharObject(strA)

// build the object that holds strB data

const bCharObject = buildCharObject(strB)

// compare number of keys in the two objects

// (anagrams must have the same number of letters)

if(Object.keys(aCharObject).length !== Object.keys(bCharObject).length) {

return false

}

// if both objects have the same number of keys

// we can be sure that at least both strings

// have the same number of characters

// Now we can compare the two objects to see if both

// have the same letters in the same amount

for(let char in aCharObject) {

if(aCharObject[char] !== bCharObject[char]) {

return false

}

}

// if both the above checks succeed,

// you have an anagram: return true

return true

}

Notice the use of [Object.keys()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys) in the snippet above. This method returns an array containing a given object’s property names or keys in the same order as they appear in the object. In this case, the array would look like this:

['f', 'i', 'n', 'd', 'r']

This is a handy way of getting an object’s properties without performing a cumbersome loop. In this challenge, you can use it together with the .length property to check if both strings have the same number of characters, which is an important feature of anagrams.

**#4 Find the Vowels**

This is probably one of the less challenging challenges (no pun intended) — in terms of difficulty — but that doesn’t detract from the fact that you could come across it during a job interview. It goes like this.

**Understanding the challenge**

You can state the vowels challenge as follows: write a function that takes a string as argument and returns the number of vowels contained in that string.

The vowels are “a”, “e”, “i”, “o”, “u”.

Examples:

findVowels('hello') // --> 2

findVowels('why') // --> 0

**Solution**

Here’s a simple iterative solution to the Vowels challenge:

const findVowels = str => {

let count = 0

const vowels = ['a', 'e', 'i', 'o', 'u']

for(let char of str.toLowerCase()) {

if(vowels.includes(char)) {

count++

}

}

return count

}

The one thing to notice here is the use of the [.includes()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) method. This method is available both on strings and arrays. You can leverage it to determine if an array contains a certain value. It returns true if the array contains the specified value and false if it doesn’t.

There’s also a more succinct solution to this challenge, which you can set up as follows:

[Report Advertisement](https://www.sitepoint.com/5-common-coding-interview-challenges/" \l "report-ad)

const findVowels = str => {

const matched = str.match(/[aeiou]/gi)

return matched ? matches.length : 0

}

The solution above revolves around the String’s [.match()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match) method. Its use in conjunction with a regular expression enables you to perform powerful searches. If the pattern of the regular expression you pass as argument to this method is found inside the specified string, the return value will be an array of the matched characters. In case no matches are found, .match() will return null.

**#5 Fibonacci**

This article couldn’t be complete without a discussion of the Fibonacci challenge, a classic question you’ll surely come across during a job interview or coding practice.

A Fibonacci sequence is an ordering of numbers where each number is the sum of the preceding two. For example, the first ten numbers of the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

**Understanding the challenge**

The Fibonacci challenge goes something like this: write a function that returns the nth entry in the Fibonacci sequence, where n is a number you pass in as argument to the function.

Example:

fibonacci(3) // --> 2

**Reasoning about the challenge**

The Fibonacci challenge involves iterating over the number of times indicated by the function’s argument and returning the entry in the Fibonacci series that you find in that position.

This way of stating the challenge immediately points to an iterative approach to solve the problem. Alternatively, you could try out a recursive solution, which would certainly impress the interviewer.

**Solution**

Here’s what the iterative solution might look like:

const fibonacci = num => {

// store the Fibonacci sequence you're going

// to generate inside an array and

// initialize the array with the first two

// numbers of the sequence

const result = [0, 1]

for(let i = 2; i <= num; i++) {

// push the sum of the two numbers

// preceding the position of i in the result array

// at the end of the result array

const prevNum1 = result[i - 1]

const prevNum2 = result[i - 2]

result.push(prevNum1 + prevNum2)

}

// return the last value in the result array

return result[num]

}

The reason why the result array above already contains the first two numbers in the series is because each entry in the Fibonacci series is made of the sum of the two preceding numbers. At the very beginning of the series there are no two numbers that you can add to produce the next number, so your loop could not possibly generate them automatically. However, you know the first two numbers are always 0 and 1, so you manually initialize your result array with those two numbers.

Let’s move on to the recursive approach:

const fibonacci = num => {

// if num is either 0 or 1 return num

if(num < 2) {

return num

}

// recursion here

return fibonacci(num - 1) + fibonacci(num - 2)

}

You keep calling fibonacci() adding up smaller and smaller numbers as arguments until you hit the base case at the top where the argument passed in is either 0 or 1.

**Conclusion**

If you’ve just gone through a few rounds of job interviews for a front-end or JavaScript developer role, especially at the junior level, you’re likely to have come across at least one or two of the questions I listed above. Even if you haven’t, you can use all of them to practice your JS coding skills.

Can you come up with alternative ways of solving the challenges? Let’s hear them!