PerformanceJS

a blog by Boris Cherny

The Best Frontend JavaScript Interview Questions (written by a Frontend Engineer)

I was at a Free Code Camp meetup in San Francisco a few days ago (for those not familiar, Free Code Camp is a group of people who get together to learn JavaScript and web development), and someone getting ready for frontend dev job interviews asked for JavaScript questions to practice. After a bit of Googling, I couldn't find any lists I could point her to and say "if you can do these, you can land a job". Some came pretty close, some were silly, but all were either incomplete or asked questions that people don't actually ask in real life.

So, based on my experiences on both sides of the interview table, here goes. These are a sampling of questions I've asked and been asked when hiring frontend engineers. Keep in mind that some places (like Google) focus more on designing efficient algorithms, so if you want to work there you should practice past CodeJam problems in addition to the stuff below. If you have a question that belongs in one of these lists (or I've made a mistake somewhere), shoot me an email.

I'll be adding/updating answers to these questions here (feel free to make PRs with additions/improvements!).

I've grouped questions into a few sections: **Concepts**, **Coding**, **Debugging**, and **System Design**:

Concepts

Be able to clearly explain these in words (no coding):

1. What is Big O notation, and why is it useful?
2. What is the DOM?
3. What is the event loop?
4. What is a closure?
5. How does prototypal inheritance work, and how is it different from classical inheritance? (this is not a useful question IMO, but a lot of people like to ask it)
6. How does `this` work?
7. What is event bubbling and how does it work? (this is also a bad question IMO, but a lot of people like to ask it too)
8. Describe a few ways to communicate between a server and a client. Describe how a few network protocols work at a high level (IP, TCP, HTTP/S/2, UDP, RTC, DNS, etc.)
9. What is REST, and why do people use it?
10. My website is slow. Walk me through diagnosing and fixing it. What are some performance optimizations people use, and when should they be used?
11. What frameworks have you used? What are the pros and cons of each? Why do people use frameworks? What kinds of problems do frameworks solve?

Coding

Implement the following functions (tests follow each question):

Easy

1. `isPrime` - Returns `true` or `false`, indicating whether the given number is prime.

```
isPrime(0)                          // false
isPrime(1)                          // false
isPrime(17)                         // true
isPrime(10000000000000)             // false
```

2. `factorial` - Returns a number that is the factorial of the given number.

```
factorial(0)                        // 1
factorial(1)                        // 1
factorial(6)                        // 720
```

3. `fib` - Returns the nth Fibonacci number.

```
fib(0)                              // 0
fib(1)                              // 1
fib(10)                             // 55
fib(20)                             // 6765
```

4. `isSorted` - Returns `true` or `false`, indicating whether the given array of numbers is sorted.

```
isSorted([])                        // true
isSorted([-Infinity, -5, 0, 3, 9])  // true
isSorted([3, 9, -3, 10])            // false
```

5. `filter` - Implement the `filter` function.

```
filter([1, 2, 3, 4], n => n < 3)    // [1, 2]
```

6. `reduce` - Implement the `reduce` function.

```
reduce([1, 2, 3, 4], (a, b) => a + b, 0) // 10
```

7. `reverse` - Reverses the given string (yes, using the built in `reverse` function is cheating).

```
reverse('')                         // ''
reverse('abcdef')                   // 'fedcba'
```

8. `indexOf` - Implement the indexOf function for arrays.

```
indexOf([1, 2, 3], 1)               // 0
indexOf([1, 2, 3], 4)               // -1
```

9. `isPalindrome` - Return `true` or `false` indicating whether the given string is a plaindrone (case and space insensitive).

```
isPalindrome('')                                // true
isPalindrome('abcdcba')                         // true
isPalindrome('abcd')                            // false
isPalindrome('A man a plan a canal Panama')     // true
```

10. `missing` - Takes an unsorted array of unique numbers (ie. no repeats) from 1 through some number *n*, and returns the missing number in the sequence (there are either no missing numbers, or exactly one missing number). Can you do it in *O(N)* time? Hint: There's a clever formula you can use.

```
missing([])                         // undefined
missing([1, 4, 3])                  // 2
missing([2, 3, 4])                  // 1
missing([5, 1, 4, 2])               // 3
missing([1, 2, 3, 4])               // undefined
```

11. `isBalanced` - Takes a string and returns `true` or `false` indicating whether its curly braces are balanced.

```
isBalanced('foo { bar { baz } boo }') // true
isBalanced('foo { bar { baz }')       // false
isBalanced('foo { bar } }')           // false
```

Intermediate

1. `fib2` - Like the `fib` function you implemented above, able to handle numbers up to `50` (hint: look up memoization).

```
fib2(0)                                 // 0
fib2(1)                                 // 1
fib2(10)                                // 55
fib2(50)                                // 12586269025
```

2. `isBalanced2` - Like the `isBalanced` function you implemented above, but supports 3 types of braces: curly `{}`, square `[]`, and round `()`. A string with interleaving braces should return false.

```
isBalanced2('(foo { bar (baz) [boo] })') // true
isBalanced2('foo { bar { baz }')         // false
isBalanced2('foo { (bar [baz] } )')      // false
```

3. `uniq` - Takes an array of numbers, and returns the unique numbers. Can you do it in *O(N)* time?

```
uniq([])                                // []
uniq([1, 4, 2, 2, 3, 4, 8])             // [1, 4, 2, 3, 8]
```

4. `intersection` - Find the intersection of two arrays. Can you do it in *O(M+N)* time (where *M* and *N* are the lengths of the two arrays)?

```
intersection([1, 5, 4, 2], [8, 91, 4, 1, 3])    // [4, 1]
intersection([1, 5, 4, 2], [7, 12])             // []
```

5. `sort` - Implement the sort function to sort an array of numbers in *O(N×log(N))* time.

```
sort([])                                // []
sort([-4, 1, Infinity, 3, 3, 0])        // [-4, 0, 1, 3, 3, Infinity]
```

6. `includes` - Return `true` or `false` indicating whether the given number appears in the given *sorted* array. Can you do it in *O(log(N))* time?

```
includes([1, 3, 8, 10], 8)              // true
includes([1, 3, 8, 8, 15], 15)          // true
includes([1, 3, 8, 10, 15], 9)          // false
```

7. `assignDeep` - Like `Object.assign`, but merges objects deeply.

```
assignDeep({ a: 1 }, {})                                        // { a: 1 }
assignDeep({ a: 1 }, { a: 2 })                                  // { a: 2 }
assignDeep({ a: 1 }, { a: { b: 2 } })                           // { a: { b: 2 } }
assignDeep({ a: { b: { c: 1 }}}, { a: { b: { d: 2 }}, e: 3 }) // { a: { b: { c: 1, d: 2 }}, e: 3 }
```

Harder

Note: For the data structures you'll implement below, the idea isn't to memorize them, but just to be able to look at the given API, Google how they work, and implement them, and to have a high level idea of what they are used for and what their tradeoffs are compared to other data structures.

1. `permute` - Return an array of strings, containing every permutation of the given string.

```
permute('')             // []
permute('abc')          // ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

2. `debounce` - Implement the debounce function.

```
let a = () => console.log('foo')
let b = debounce(a, 100)
b()
b()
b() // only this call should invoke a()
```

3. `seq` - Resolve an array of promises in sequence (as opposed to `Promise.all`, which does it in parallel).

```
let a = Promise.resolve('a')
let b = Promise.resolve('b')
let c = Promise.resolve('c')
await seq([a, b, c])                    // ['a', 'b', 'c']
await seq([a, c, b])                    // ['a', 'c', 'b']
```

4. Implement a `LinkedList` class without using JavaScript's built-in arrays (`[]`). Your LinkedList should support just 2 methods: `add` and `has`:

```
class LinkedList {...}
let list = new LinkedList(1, 2, 3)
list.add(4)                          // undefined
list.add(5)                          // undefined
list.has(1)                          // true
list.has(4)                          // true
list.has(6)                          // false
```

5. Implement a `HashMap` class without using JavaScript's built-in objects (`{}`) or `Maps`. You are provided a `hash()` function that takes a string and returns a number (the numbers are mostly unique, *but sometimes two different strings will return the same number*):

```
function hash (string) {
  return string
    .split('')
    .reduce((a, b) => ((a << 5) + a) + b.charCodeAt(0), 5381)
}
```

   Your HashMap should support just 2 methods, `get`, `set`:

```
let map = new HashMap
map.set('abc', 123)                  // undefined
map.set('foo', 'bar')                // undefined
map.set('foo', 'baz')                // undefined
map.get('abc')                       // 123
map.get('foo')                       // 'baz'
map.get('def')                       // undefined
```

6. Implement a `BinarySearchTree` class. It should support 4 methods: `add`, `has`, `remove`, and `size`:

```
let tree = new BinarySearchTree
tree.add(1, 2, 3, 4)
tree.add(5)
tree.has(2)                          // true
tree.has(5)                          // true
tree.remove(3)                       // undefined
tree.size()                          // 3
```

7. Implement a `BinaryTree` class with breadth first search and inorder, preorder, and postorder depth first search functions.

```
let tree = new BinaryTree
let fn = value => console.log(value)
tree.add(1, 2, 3, 4)
tree.bfs(fn)                         // undefined
tree.inorder(fn)                     // undefined
tree.preorder(fn)                    // undefined
tree.postorder(fn)                   // undefined
```

Debugging

For each of the following questions, start by understanding and explaining why the given piece of code doesn't work. Then propose a couple of fixes, and rewrite the code to implement one of the fixes you proposed so the program works correctly:

1. I want this code to log out `"hey amy"`, but it logs out `"hey arnold"` - why?

```
function greet(person) {
  if (person == { name: 'amy' }) {
    return 'hey amy'
  } else {
    return 'hey arnold'
  }
}
greet({ name: 'amy' })
```

2. I want this code to log out the numbers `0`, `1`, `2`, `3` in that order, but it doesn't do what I expect (this is a bug you run into once in a while, and some people love to ask about it in interviews).

```
for (var i = 0; i < 4; i++) {
  setTimeout(() => console.log(i), 0)
```

```
}
```

3. I want this code to log out `"doggo"`, but it logs out `undefined`!

```
let dog = {
  name: 'doggo',
  sayName() {
    console.log(this.name)
  }
}
let sayName = dog.sayName
sayName()
```

4. I want my dog to `bark()`, but instead I get an error. Why?

```
function Dog(name) {
  this.name = name
}
Dog.bark = function() {
  console.log(this.name + ' says woof')
}
let fido = new Dog('fido')
fido.bark()
```

5. Why does this code return the results that it does?

```
function isBig(thing) {
  if (thing == 0 || thing == 1 || thing == 2) {
    return false
  }
  return true
}
isBig(1)    // false
isBig([2])  // false
isBig([3])  // true
```

System design

If you're not sure what "system design" means, start here.

1. Talk me through a full stack implemention of an autocomplete widget. A user can type text into it, and get back results from a server.
    - How would you design a frontend to support the following features:
        - Fetch data from a backend API
        - Render results as a tree (items can have parents/children - it's not just a flat list)
        - Support for checkbox, radio button, icon, and regular list items - items come in many forms
    - What does the component's API look like?
    - What does the backend API look like?
    - What perf considerations are there for complete-as-you-type behavior? Are there any edge cases (for example, if the user types fast and the network is slow)?
    - How would you design the network stack and backend in support of fast performance: how do your client/server communicate? How is your data stored on the backend? How do these approaches scale to lots of data and lots of clients?
2. Talk me through a full stack Twitter implementation (shamelessly stolen from my friend Michael Vu).
    - How do you fetch and render tweets?
    - How do you update tweets as new ones come in? How do you know when new ones came in?
    - How do you search tweets? How do you search by author? Talk me through your database, backend, and API designs.

Further resources

If you got this far and want more, below are some high quality resources that I've found helpful.

For more things to implement:

- The Algorithm Design Manual

- Past CodeJam problems
- keon/algorithms

For some good reading:

- The Algorithm Design Manual
- JavaScript Allonge
- You Don't Know JS
- Effective JavaScript

View repo on GitHub ➜frontend-interview-questionsPosted...

- By Boris Cherny
- 21 days ago
- In JSFunctional Programming

Subscribe