Versatile Add

We'll cover the concepts of scope, closures, callbacks, and value vs. reference in this unique problem. This problem tests your understanding of the core concepts of JavaScript.

Introduction

In this problem, we'll write a simple function that adds two numbers. The challenge present is that this function will have to be extremely versatile.

This problem will test your understanding of:

- scope
- callbacks
- closures
- value vs. reference

If you understand these concepts well, this problem can still be extremely difficult, but you have the ability to solve it.

Let's get started.

Instructions

Write a function that adds 2 numbers. It should work as follows:

```
add(3, 4); // -> 7
add(10, 12); // -> 22
```

However, it should also work as follows:

```
add(3)(4); // -> 7
add(10)(12); // -> 22
```

BONUS:

Make the following lines of code also function properly.

```
add(3)()(4); // -> 7
add(3)()()()(4); // -> 7
add(10)()()()()()()()()()()()(12); // -> 22

add()(3)(4); // -> 7
add()()()()(10)(12); // -> 22

add()(3)()(4); // -> 7
add()()()()()(10)()()()(12); // -> 22
```

Hints

- Start with what you know and work down the examples.

```
function add(num1, num2) {
    // Your code here
}
```

Solution

1

We're going to start with the simplest version of our function. We want these lines to work correctly.

```
add(3, 4); // -> 7
add(10, 12); // -> 22
```

We need to write a function that receives two numbers as arguments and return the sum.

```
function add(num1, num2) {
    return num1 + num2;
}
```

Simple enough. Now the hard part starts.

2

We'll focus on getting these two lines to work.

```
add(3)(4); // -> 7
add(10)(12); // -> 22
```

We can see that if two arguments are provided, the function should return the sum. If one argument is provided, the function needs to do something else.

To detect if a second argument is passed in, we need to check if it's `undefined`.

```
function add(num1, num2) {
    if(num2 === undefined) {
        // return something else
    }

    return num1 + num2;
}
```

What do we need to return on line 3 above? Let's take a look at a sample again.

```
add(3)(4); // -> 7
```

Another way of writing this is:

```
const intermediate = add(3);
intermediate(4); // -> 7
```

It's clear that `intermediate` here needs to be a function, so we'll need to return a function on line 3. That function needs to accept an argument.

```
1  function add(num1, num2) {
2      if(num2 === undefined) {
3          return function(num3) {
4              // return something here
5          }
6      }
7
8      return num1 + num2;
9  }
```

We need the function that we return to accept an argument (`num3`) and return the sum of that number plus the original argument (`num1`).

Due to the rules of scope in JavaScript, the anonymous function will retain access to the variable `num1` . It can be used inside the new function.

```
1  function add(num1, num2) {
2      if(num2 === undefined) {
3          return function(num3) {
4              return num1 + num3;
5          }
6      }
7
8      return num1 + num2;
9  }
```

Show Results Show Console

1.46s

4 of 4 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	add(3, 4)	7	7	Succeeded
✓	add(5, 10)	15	15	Succeeded
✓	add(3)(4)	7	7	Succeeded
✓	add(5)(10)	15	15	Succeeded

That's it. We've made those examples function correctly.

3 - Bonus

We need the next set of the samples to work now.

```
add(3)()()()(4); // -> 7
```

Again, let's rewrite the first line above.

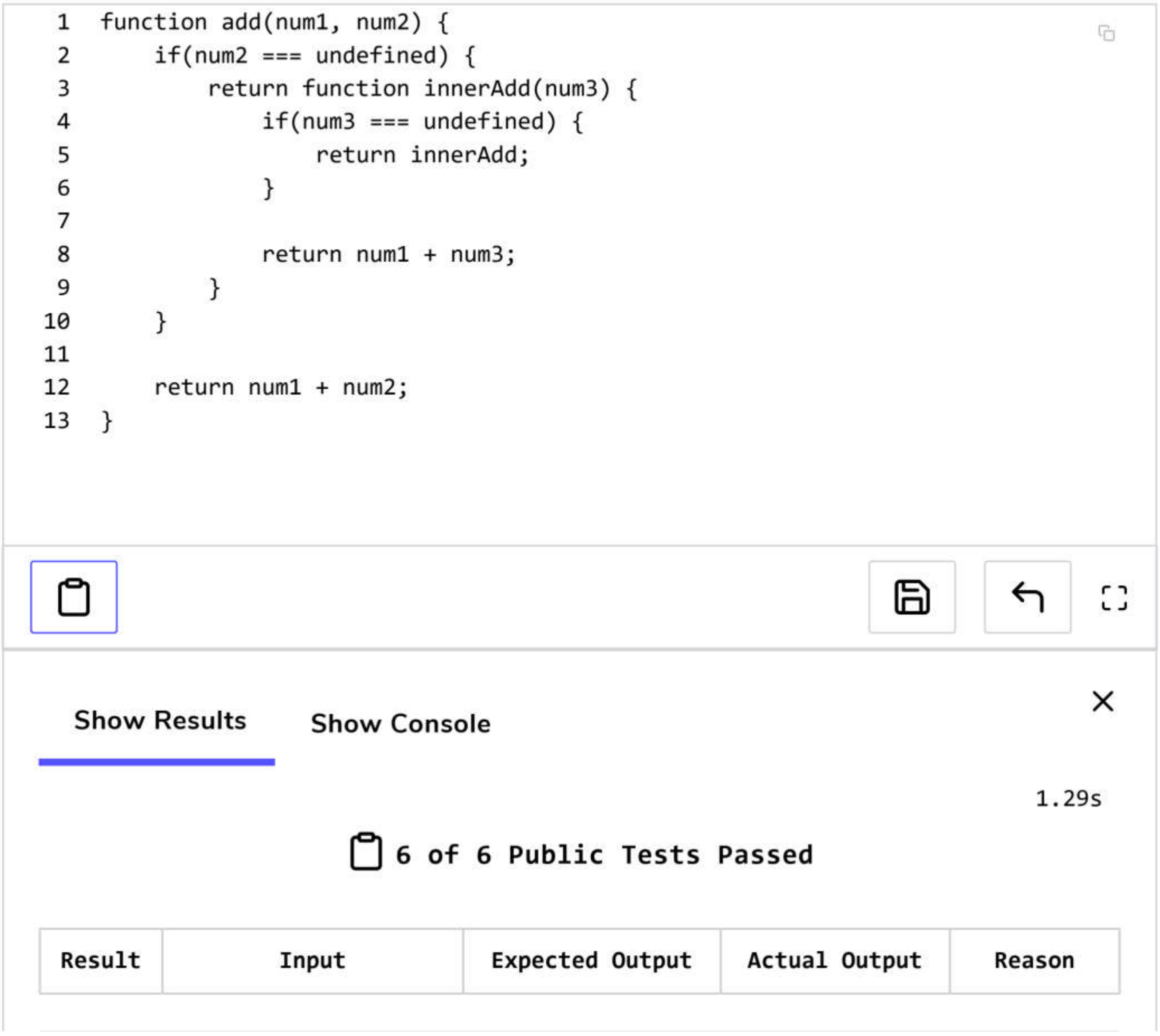
```
let intermediate = add(3);
intermediate();
intermediate();
intermediate();
intermediate(4); // -> 7
```

In this case, we're supplying a single argument to `add`. Our function `add` needs to return a function that accepts 0 or 1 arguments.

If we provide no arguments to the inner function above (`intermediate`), it should return a function that *indefinitely* waits for an argument. It should be infinitely invokable, returning the correct value when provided an argument.

In other words, if we provide no input, the inner function has to return a function that does the same exact thing every time. It has to return *the same function* every time.

It has to return itself. Let's add this change in to our function.



```
function add(num1, num2) {
    if(num2 === undefined) {
        return function innerAdd(num3) {
            if(num3 === undefined) {
                return innerAdd;
            }

            return num1 + num3;
        }
    }

    return num1 + num2;
}
```

Show Results Show Console

1.29s

6 of 6 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason

Result	Input	Expected Output	Actual Output	Reason
✓	add(3, 4)	7	7	Succeeded
✓	add(5, 10)	15	15	Succeeded
✓	add(3)(4)	7	7	Succeeded
✓	add(5)(10)	15	15	Succeeded
✓	add(3)()(4)	7	7	Succeeded
✓	add(5)()()()()(10)	15	15	Succeeded

Phew. One more part.

4 - Bonus

We need the last few examples working.

```
add()(3)()(4); // -> 7
add()()()()()(10)()()()(12); // -> 22
```

The only thing that changes here is that we should be able to invoke `add` indefinitely, just as we can invoke `innerAdd` indefinitely. The solution will be very similar to what we've already implemented.

(/learn)

```
function add(num1, num2) {
    if(num1 === undefined) {
        return add;
    }

    if(num2 === undefined) {
        return function innerAdd(num3) {
            if(num3 === undefined) {
                return innerAdd;
            }

            return num1 + num3;
        }
    }

    return num1 + num2;
}
```

That’s the whole function.

Conclusion

This is by far one of the most challenging problems present in this set. It requires deep knowledge of fundamental JavaScript concepts, primarily scope and the idea of variables containing references.

← Back

Function Bind

Next →

Congratulations

✓ Mark as Completed

Report an Issue