LEARN RECURSION BY BUILDING A DECIMAL TO BINARY CONVERTER

Introduction:

Recursion is a programming concept where a function calls itself. This can reduce a complex problem into simpler sub-problems, until they become straightforward to solve.

In this project, you'll build a decimal-to-binary converter using JavaScript. You'll learn the fundamental concepts of recursion, explore the call stack, and build out a visual representation of the recursion process through an animation.

Step 1:

In this project, you'll build a decimal and binary converter and learn about both number systems. You'll also learn about recursion by using it to perform the conversions.

All of the HTML and CSS for this project has been provided for you.

When you're ready to get started, use the `.getElementById()` method to get the `input` element with the id `"number-input"`, and store it in a variable called `numberInput`. Use the same method to get the `button` element with the id `"convert-btn"` and store it in a variable called `convertBtn`, and the `output` element with the id `"result"` and store it in a variable called `result`.

NOTE: This project will only convert positive numbers into binary.

Step 2:

Now you'll do some setup to check the value in the number input element whenever the user clicks the `Convert` button.

First, create an empty function called checkUserInput.

Step 3:

A good way to test that everything is working is to log the value attribute of numberInput to the console. As a reminder, you can access the value attribute of an element by using dot or bracket notation.

Within the checkUserInput function, use console.log() to log the value of numberInput to the console.

Step 4:

Now that your checkUserInput() function is set up for testing, you can use an event listener to call the function when users click the Convert button.

Chain the .addEventListener() method to convertBtn. The event listener should listen for click events and take a reference to the checkUserInput function as a callback. Remember that function references are not called with parentheses.

Once that's done, whenever you click the Convert button, the value of the number input should be logged to the console.

Step 5:

Your Convert button should be working now. But it could get tiring for users to enter in a number, then click that button each time they want to convert from decimal to binary. It would be much more convenient to perform the conversion when the Enter or Return key is pressed.

The keydown event fires every time a user presses a key on their keyboard, and is a good way to add more interactivity to input elements.

Chain `.addEventListener()` to `numberInput`. The event listener should listen for `keydown` events and take an empty arrow function as a callback.

Step 6:

Whenever an event listener is triggered by an event, an event object is created automatically. You don't always need to use this object, like with your `click` handler above, but it can be useful to access information about the event that was triggered.

First, pass `e` as a parameter to your callback function. Remember that `e` is a common parameter name for the event object. Next, log `e` to the console in the body of your callback function.

Step 7:

If you open your browser's console and type in the number input, you'll see event objects logged to the browser. And if you take a closer look at one of those event objects, you'll see helpful properties like `type` and `target`.

Since you want to perform an action when the `Enter` key is pressed, the most helpful property is `key`, which tells you the string value of the key that was pressed.

Remove the `console.log()` statement from the callback function and add an `if` statement that checks if `e.key` is equal to the string `"Enter"`. Leave the body of your `if` statement empty for now.

Note: Since the `Enter` and `Return` keys have similar functions, they both have the same string value of `"Enter"`.

Step 8:

Next, within the body of the `if` statement, call the `checkUserInput()` function. After this, if you enter numbers into the number input and press the `Enter` / `Return` key, you should see numbers logged to the console.

Step 9:

Now that your `Convert` button and number input are listening for clicks and `Enter` key presses, it's time to complete the `checkUserInput()` function.

It would be helpful to alert users if they don't enter a value into the number input, or the number they enter is invalid. While the `input type="number"` element makes validation easier by only allowing numbers and some special characters, remember that all values you get from HTML elements are actually strings. Also, if the number input is empty, the `value` property will be an empty string.

Inside your `checkUserInput` function, use an `if` statement to check if the `value` of `numberInput` is equal to an empty string. Leave the body of the `if` statement empty for now.

Step 10:

In an earlier project you learned about truthy and falsy values, which are values that evaluate to `true` or `false`. In JavaScript, some common falsy values you'll see are `null`, `undefined`, the number `0`, and empty strings.

Rather than check if a value is equal to a falsy value, you can use the logical NOT operator (`!`) to check if the value itself is falsy. For example:

Example Code:

```
const num = 0;
```

```
console.log(num === 0); // true

console.log(!num); // true
```

Update the condition in your `if` statement to use the logical NOT operator to check if `numberInput.value` is falsy.

Step 11:

Because the `input type="number"` element allows special characters like `.`, `+`, and `e`, users can input floats like `2.2`, equations like `2e+3`, or even just `e`, which you don't want to allow.

A good way to check and normalize numbers in JavaScript is to use the built-in `parseInt()` function, which converts a string into an integer or whole number. `parseInt()` takes at least one argument, a string to be converted into an integer, and returns either an integer or `NaN` which stands for `Not a Number`. For example:

Example Code

```
parseInt(2.2); // 2

parseInt("2e+3"); // 2

parseInt("e") // NaN
```

Add a logical OR operator (`||`) after the first condition in your `if` statement. Then, pass the value of `numberInput` into the `parseInt()` function as the second condition of your `if` statement.

Step 12:

Next, you need to check if the value returned by the `parseInt()` function is a number or not.

To do that, you can use the isNaN() function. This function takes in a string or number as an argument, and returns true if it evaluates to NaN. For example:

Example Code:

```
isNaN("test"); // true

isNaN(2); // false

isNaN("3.5"); // false
```

Update the second condition in your if statement to use the isNaN() function to check if the value returned by parseInt() is NaN.

Also, as we mentioned in step 1 that we are considering only positive numbers, we should add a third condition in the if statement to check whether the number is less than 0 (i.e negative numbers).

Example Code:

```
6 < 0; // false

-1 < 0; // true

-8 < 0; // true

 0 < 0; //false
```

Step 13:

Now you can alert the user if they don't enter a number, or the number is invalid before you attempt to convert it into binary.

In the body of the if statement, use the alert() method to display the text "Please provide a decimal number greater than or equal to 0".

Note that alert() is a method on the window object in the browser, so you can use either window.alert() or alert().

Step 14:

After alerting the user if the number input is empty or the value is not a number, you can use the `return` keyword to break out of this function early. This will prevent future code in this function from running.

Add the `return` keyword after `alert()`.

Step 15:

Now you'll start building the function to actually do the decimal to binary conversion.

Create a function called `decimalToBinary` with `input` as a parameter. Leave the body of the function empty for now.

Step 16:

Within your `checkUserInput` function, remove the `console.log()` statement. Then, call the `decimalToBinary` function and pass in the `value` of `numberInput` as an argument. Also, make sure to use the `parseInt()` function to convert the input into a number.

Step 17:

Finally, you should clear the number input by setting its value to an empty string. Then later when you convert several numbers in a row, you won't have to delete the previous number before entering the next one.

Set the `value` property of `numberInput` to an empty string.

Step 18:

Now that your function is set up, it's time to learn about binary numbers.

Binary numbers are a base-2 number system. Unlike the base-10 or decimal number system we use every day that uses 10 digits (0-9) to form numbers, the binary number system only has two digits, 0 and 1. In computer science, these binary digits are called bits, and are the smallest unit of data computers can process. For computers, 0 represents false or "off", and 1 represents true or "on".

In your decimalToBinary function, use the return keyword to return a string of the binary number representation of true.

Note: Binary numbers can be long sequences that start with 0, so they are often represented as strings.

Step 19:

In the base-2 number system, the rightmost digit represents the ones place, the next digit to the left represents the twos place, then the fours place, then the eights place, and so on. In this system, each digit's place value is two times greater than the digit to its right.

Here are numbers zero to nine in the base-10 and base-2 number systems:

Example Code:

Base-10	Base-2
0	0
1	1
2	10

3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Notice that binary numbers are formed from left to right, from the digit with the greatest place value on the left, to the least significant on the right. For example, the number 3 in binary is 11, or 1 in the twos place and 1 in the ones place. Then for the number 4, a digit to represent the fours place is included on the left and set to 1, the twos place is 0, and the ones place is 0.

In your decimalToBinary function, convert the number 10 into binary and return it as a string.

Step 20:

Bits are often grouped into an octet, which is an 8-bit set known as a byte. A byte can represent any number between 0 and 255. Here are the placement values for each bit in a byte:

Example Code:

```
128 | 64 | 32 | 16 | 8 | 4 | 2 | 1
```

Because bits are often grouped into bytes, it's common to see binary numbers represented in groups of eight, sometimes with leading zeros. For example, the number 52 can be represented as 110100, or 00110100 with leading zeros. Here's how that breaks down with the placement values:

Example Code:

```
128 | 64 | 32 | 16 | 8 | 4 | 2 | 1

  0 |  0 |  1 |  1 | 0 | 1 | 0 | 0
```

In your `decimalToBinary` function, convert the number `118` into binary with leading zeros and `return` it as a string.

Step 21:

Now that you're familiar with binary numbers, it's time to finish building the function to do the conversion for you. You'll start off with a simpler solution first, then refactor that into a recursive solution.

First, you need to create some arrays to store the inputs and results of the division you'll do in the following steps. These will make it easier to see how the decimal to binary conversion works.

Remove the `return` statement from your `decimalToBinary` function. Then, declare variables named `inputs`, `quotients`, and `remainders`, and assign an empty array to each of them.

Step 22:

Set `input` equal to the number `0` for now. We'll change this in the next few steps.

Step 23:

For the decimal to binary conversion, you need to divide `input` by `2` until the quotient, or the result of dividing two numbers, is `0`. But since you don't know how many times you need to divide `input` by `2`, you can use a `while` loop to run a block of code as long as `input` is greater than `0` and can be divided.

As a reminder, a `while` loop is used to run a block of code as long as the condition evaluates to `true`, and the condition is checked before the code block is executed. For example:

Example Code:

```
let i = 0;

while (i < 5) {

  console.log(i);

  i++;

}
```

Create a `while` loop that runs as long as `input` is greater than `0`. Leave the body of the loop empty for now.

Step 24:

The tricky part about `while` loops is that, if you're not careful, they can run forever. This is called an infinite loop, and can cause your browser to crash.

To avoid infinite loops, you need to make sure that the condition for the `while` loop eventually becomes `false`. In this case, you want to make sure that the `input` variable eventually becomes `0`.

Move the `input = 0` statement into the body of the `while` loop. This will make it so that the loop will only run up to one time.

Step 25:

To divide numbers in JavaScript, use the division operator (`/`). For example:

Example Code:

```
const quotient = 5 / 2; // 2.5
```

In the example above, `5` is the dividend, or the number to be divided, and `2` is the divisor, or the number to divide by. The result, `2.5`, is called the quotient.

Inside your `while` loop, create a variable named `quotient` and assign it the value of `input` divided by `2`.

Step 26:

Like you saw in the last step, division can lead to a floating point number, or a number with a decimal point. The best way to handle this is to round down to the nearest whole number.

Use the `Math.floor()` function to round down the quotient of `input` divided by `2` before it's assigned to `quotient`.

Step 27:

Now that you have an operation that will lower the value of `input` each time the loop runs, you don't have to worry about the loop running forever.

Update the last line of your `while` loop and assign `quotient` to `input`.

Step 28:

Next, you need to calculate the remainder of `input` divided by `2`. You can do this by using the remainder operator (`%`), which returns the remainder of the division of two numbers. For example:

Example Code:

```
const remainder = 5 % 2; // 1
```

In other words, the dividend, `5`, can be divided by the divisor, `2`, multiple times. Then you're left with a remainder of `1`.

Inside your `while` loop, create a variable named `remainder` and use the remainder operator to assign it the remainder of `input` divided by `2`.

Step 29:

Inside your `while` loop, use the `.push()` method to append `input` to the `inputs` array. This will help you get a better idea of how the conversion works later when you log the contents of your arrays to the console.

Step 30:

Use `.push()` to append the `quotient` variable to the `quotients` array. Also, append the `remainder` variable to the `remainders` array.

Step 31:

Now's a good time to check your work.

Log the text `"Inputs: "`, followed by a comma, followed by the `inputs` array to the console.

Step 32:

Next, log the text `"Quotients: "`, followed by a comma, followed by the `quotients` array to the console. Also, log the text `"Remainders: "`, followed by a comma, followed by the `remainders` array.

Step 33:

Now if you enter in the number `6` and click the `Convert` button, you'll see the following output:

Example Code:

```
Inputs:  [ 6, 3, 1 ]

Quotients:  [ 3, 1, 0 ]

Remainders:  [ 0, 1, 1 ]
```

Notice that the `remainders` array is the binary representation of the number `6`, but in reverse order.

Use the `.reverse()` method to reverse the order of the `remainders` array, and `.join()` with an empty string as a separator to join the elements into a binary number string. Then, set `result.innerText` equal to the binary number string.

Step 34:

Your `decimalToBinary` function works well, but there is an issue – because of the condition in your `while` loop, it only works for numbers greater than `0`. If you try to convert `0` to binary, nothing will get added to the page.

To fix this, add an `if` statement to check if `input` is equal to `0`. Leave the body of the `if` statement empty for now.

Step 35:

Within the body of the if statement, set the innerText property of result equal to the string "0". Then, use an early return statement to break out of the function early.

Step 36:

Now your decimalToBinary function is complete. Feel free to play around with it.

But there are some ways to improve it. For example, it's not necessary to keep track of the inputs and quotients. We can clean things up so the function is more efficient.

First, remove everything in the body of the decimalToBinary function. Then, use let to create a variable named binary and assign it an empty string.

Step 37:

Since you'll want to display the result of the conversion, assign the binary variable to the innerText property of result at the end of the function.

Step 38:

Create a while loop that runs as long as input is greater than 0. Inside the loop, assign 0 to input for now.

Note: Be careful not to trigger the decimalToBinary function before you set input equal to 0 inside the loop. Otherwise, you could cause an infinite loop.

Step 39:

Recall that, each time the loop runs, `input` is the quotient of the previous value of `input` divided by `2`, rounded down. Eventually, `input` is less than `1`, and the loop stops running.

You can do this in a single step.

Inside your `while` loop, set `input` equal to the quotient of `input` divided by `2`. Also, remember to use `Math.floor()` to round the quotient down.

Step 40:

In the previous version of this function, you pushed the remainder of `input` divided by `2` to `binaryArray`. Then later you reversed and joined the entries into a binary number string.

But it would be easier to use string concatenation within the loop to build the binary string from right to left, so you won't need to reverse it later.

First, use the `remainder` operator (`%`) to set `binary` equal to the remainder of `input` divided by `2`.

Step 41:

Then, use the addition operator to add the current value of `binary` to the end of the equation `input % 2`. This is what will build the binary string from right to left.

Step 42:

To clean things up a bit, wrap `input % 2` in parentheses. This can sometimes change the order of operations, but in this case, it just makes your code easier to read.

Step 43:

Finally, you need to handle cases where `input` is `0`. Rather than update the DOM and return early like you did before, you can update the `binary` string and let the rest of the code in the function run.

Create an `if` statement that checks if `input` is equal to `0`. If it is, set `binary` equal to the string `"0"`.

Step 44:

Awesome. Now you have a more efficient way to convert decimal numbers into binary. After learning a bit about the call stack and recursion, you'll refactor the decimalToBinary function to use recursion instead of a while loop.

Create a function named a that returns the following: "freeCodeCamp " + b().

Step 45:

Next, create a function named `b` that returns the following: `"is " + c()`.

Also, create a function named `c` that returns the following: `"awesome!"`.

Step 46:

Finally, call a() from within a console.log() statement to log the output to the console.

Step 47:

In computer science, a stack is a data structure where items are stored in a LIFO (last-in-first-out) manner. If you imagine a stack of books, the last book you add to the stack is the first book you can take off the stack. Or an array where you can only .push() and .pop() elements.

The call stack is a collection of function calls stored in a stack structure. When you call a function, it is added to the top of the stack, and when it returns, it is removed from the top / end of the stack.

You'll see this in action by creating mock call stack.

Initialize a variable named callStack and assign it an empty array.

Step 48:

When your code runs, the a() function is added to the call stack first.

In your callStack array, add the following string: 'a(): returns "freeCodeCamp " + b()'. This represents the function call and the code that will be executed.

Note: Since the string you're adding includes double quotation marks ("), wrap it in single quotation marks (') or backticks (`).

Step 49:

Then, since a() calls b(), the function b() is added to the call stack.

Next, add the following string to your callStack array: "b(): returns 'is ' + c()".

Remember that the call stack is a LIFO data structure, so the last function is added to the top or end of the stack, similar to pushing an element into an array.

Step 50:

And since b() calls c(), the function c() is added to the call stack.

Add the following string to your callStack array: "c(): returns 'awesome!'".

Step 51:

Your call stack is complete. As you can see, a() is at the bottom or beginning of the stack, which calls b() in the middle, which calls c() at the top or end. Once they're all in place, they begin to execute from top to bottom.

c() executes, returns the string "awesome!", and is popped off or removed from the top of the stack.

Remove your "c(): returns 'awesome!'" string from the top of the callStack array.

Step 52:

Then the function b() executes and evaluates to "is " + "awesome!".

Update your mock call to b() so it looks like this: "b(): returns 'is ' + 'awesome!'".

Step 53:

Now that b() has executed, pop it off the call stack. Then, update your mock call to a() to the following: "a(): returns 'freeCodeCamp ' + 'is awesome!'".

Step 54:

Finally, a() returns the concatenated string "freeCodeCamp is awesome!".

Pop a() off the top of the call stack.

Step 55:

While that's a simple example, it demonstrates how the call stack steps through your code and calls multiple functions.

Now it's time to jump into recursion, and see how the call stack fits into the picture.

Remove your callStack array, the a(), b(), and c() functions, and the console.log() statement.

Step 56:

Now you'll create a function that will count down from a given number to zero using recursion.

Create a new function named countdown with number as a parameter. Leave the function body empty for now.

Step 57:

The first thing you need to do is log the current value of number to the console to act as the countdown.

Use `console.log()` to log `number` to the console.

Step 58:

A recursive function is a function that calls itself over and over. But you have to be careful because you can easily create an infinite loop. That's where the base case comes in. The base case is when the function stops calling itself, and it is a good idea to write it first.

Since your `countdown()` function will count down from a given number to zero, the base case is when the `number` parameter is equal to `0`. Then it should `return` to break out of its recursive loop.

Use an `if` statement to check if `number` is equal to `0`. If it is, use the `return` keyword to break out of the function.

Step 59:

Recursive functions also have a recursive case, which is where the function calls itself.

First, convert your `if` statement into an `if...else` statement. Leave the body of your `else` statement empty for now.

Step 60:

When writing the recursive case, you need to remember two things:

1. What is the base case?
2. What is the least amount of work you need to do to get closer to the base case?

Since the base case is when `number` is equal to `0`, you need to call `countdown()` again while also lowering the value of `number` by `1`.

Inside the `else` block, call `countdown()` and pass it `number - 1` as an argument.

Step 61:

It's time to test your function. Call `countdown()` with an argument of `3` to see if it works.

Step 62:

To really see the call stack in action, you just need to modify the function slightly.

First, rename the `countdown()` function to `countDownAndUp()`. Remember to update your function calls, too.

Step 63:

In your base case, log `"Reached base case"` to the console.

Step 64:

Then, log `number` to the console after your recursive `countDownAndUp(number - 1)` function call.

Step 65:

Now you should see a countdown from `3` to `0`, followed by `Reached base case`, and a count from `1` to `3`. This is because, after the recursive loop is finished, the function will continue to execute the code after the recursive call. This is why you see `Reached base case` before the count from `1` to `3`.

Now that you have a better understanding of how the call stack and recursion work, you'll refactor the decimalToBinary() function to use recursion instead of a while loop.

First, remove your countDownAndUp() function and function call.

Step 66:

Then, remove the contents of your decimalToBinary() function. Leave the body of the function empty for now.

Step 67:

As a reminder, it's often best to start with the base case when writing a recursive function so you know what you're working towards, and to prevent an infinite loop.

Similar to your last implementation, you'll divide input by 2 repeatedly until input is 0.

Use an if statement to check if input is equal to 0. If it is, return an empty string.

Step 68:

For the recursive case, add an else statement and return the result of calling decimalToBinary(). Pass in input divided by 2 rounded down with Math.floor() as the argument.

Step 69:

This effectively lowers the input by roughly half each time the decimalToBinary() function is called.

However, remember that the binary number string is built by calculating the remainder of input divided by 2 and concatenating that to the end.

After your call to decimalToBinary(), use the addition operator (+) to concatenate the remainder of input divided by 2 to the end of the string your recursive function returns. Also, wrap the operation in parentheses.

Step 70:

Finally, in your checkUserInput() function, set the textContent property of result equal to the string returned by your decimalToBinary() function.

Step 71:

Your converter should be working now. Feel free to try out different numbers and think about what is happening each time decimalToBinary() is called.

But if you test your converter with 0, you'll see that nothing happens. This is because you return an empty string in your base case when input is 0. We can fix this now.

Update your base case so that it returns the string "0" when input is equal to 0.

Step 72:

This mostly works – if you convert 0 into binary, 0 is displayed on the page. But now when you convert other numbers greater than 0, your binary number starts with a leading 0. For example, if you convert 1, the result is 01.

But if you think about it, `0` and `1` in base-10 always convert to `0` and `1` in base-2, respectively. So you can add another base case to handle when `input` is `1`.

Add an `else if` statement to your function that checks if `input` is equal to `1`. If it is, return the string `"1"`.

Step 73:

Now everything should work as expected. And since you know that `input` will either be the numbers `0` or `1` at this point, you can combine your two base cases and just return `input` as a string.

For a reliable way to convert a value into a string, even falsy values like `null` and `undefined`, you can use the `String()` function. For example:

Example Code:

```
const num = 5;

console.log(String(num)); // "5"

console.log(String(null)); // "null"
```

Combine your `if` and `else if` statements into a single `if` statement checking if `input` is equal to `0` or `1`. If it is, use the `String()` function to convert `input` into a string and return it.

Step 74:

Now your `decimalToBinary()` function is complete. Feel free to test it out.

If you're still confused about how it works under the hood, don't worry. Next, you'll create a simple animation to help you understand what's happening each step of the way.

Create a new function called showAnimation. Leave the body of the function empty for now.

Step 75:

You'll show the animation when users try to convert the decimal number 5 to binary, so you'll need to add a check for that within your checkUserInput() function.

Use an if statement to check if the value attribute of numberInput is equal to the number 5. Remember to use the parseInt() function to convert the string into a number before comparing it to 5. Leave the if statement empty for now.

Step 76:

If the value of numberInput is equal to 5, call the showAnimation() function, then return early.

Step 77:

Now your showAnimation() function is set up. But if you look at your checkUserInput() function, you'll notice that it is calling parseInt() to convert numberInput.value into a number several times.

This is generally a poor practice, for reasons like performance concerns or even just the fact that you'd have to change your logic in multiple places to update the parseInt() call.

To fix this, create a new variable to store the converted number. Then you only have to convert the number once and can use it throughout the function.

Create a new variable called `inputInt` and assign it the number converted from `numberInput.value`.

Step 78:

Replace all instances of `parseInt(numberInput.value)` with `inputInt`.

Step 79:

Now that your showAnimation() function is set up, let's do some testing.

Add three console.log() statements in the showAnimation() function to log the text "free", "Code", and "Camp" to the console. You should see that text in the console when you enter 5 into the number input and click the Convert button.

Step 80:

Before you start writing code for the animation, let's take a look at the function you'll use to add and remove elements from the DOM: `setTimeout`.

The `setTimeout` function takes two arguments: a callback function and a number representing the time in milliseconds to wait before executing the callback function.

For example, if you wanted to log `"Hello, world!"` to the console after `3` seconds, you would write:

Example Code:

```
setTimeout(() => {

  console.log("Hello, world!");
}, 3000);
```

Use the `setTimeout` function to add a one second delay before the text `"Code"` is logged to the console. Then see what happens after you enter `5` into the number input and click the `Convert` button.

Step 81:

If you test your code, you'll notice that your console logs are not in the expected order. Instead of logging `"free"`, pausing for a second before logging `"Code"`, and finally logging `"Camp"`, you'll see this:

Example Code:

```
free

Camp

Code
```

This is because the `setTimeout()` function is asynchronous, meaning that it doesn't stop the execution of the rest of your code. All the code in the `showAnimation()` function runs line by line, but because `setTimeout()` is asynchronous, `free` and `Camp` are logged to the console immediately, and then `Code` is logged to the console after a one second delay.

One way to fix this is to use multiple `setTimeout()` functions. Use `setTimeout()` to log `free` to the console after half a second, or `500` milliseconds.

Step 82:

While asynchronous, or async, code can be difficult to understand at first, it has many advantages. One of the most important is that it allows you to write non-blocking code.

For example, imagine you're baking a cake, and you put the cake in the oven and set a timer. You don't have to sit in front of the oven waiting the entire time – you can wash dishes, read a book, or do anything else while you wait for the timer to go off.

Async code works in a similar way. You can start an async operation and other parts of your code will still work while that operation is running.

You'll learn more about async code in future projects, but the `setTimeout()` function is a good introduction.

Add a `1500` millisecond delay before the text `"Camp"` is logged to the console.

Step 83:

Now you're ready to start on the animation itself. You'll use an array of objects to store data for each frame of the animation.

First, create a new variable called `animationData` and assign it an empty array.

Step 84:

Next, you'll create an object to represent the first frame of your animation. Your object should have two properties or keys: `inputVal`, and `addElDelay`.

`inputVal` will represent the value of the input each time your recursive function runs. And `addElDelay` will be the delay between adding DOM elements to the page.

Add an object to animationData with an inputVal property set to 5, and an addElDelay property set to 1000.

Step 85:

Recall that the call stack is a LIFO (last in, first out) data structure. This means that, as functions are called, they are added to the top or end of the stack, and as functions return, they are removed from the top of the stack.

Treat your animationData array as a stack and add a new object to it. Your new object should have the properties inputVal, and addElDelay set to 2, and 1500, respectively. Remember to add this object to the top of the stack, or in other words, to the end of the animationData array.

Step 86:

Add another object to the animationData array. Your new object should have the properties inputVal, and addElDelay set to 1, and 2000, respectively. Remember to treat the animationData array as a stack and add the new object to the top of the stack.

Step 87:

Now you'll start building the animation itself.

First, use the document.getElementById() method to select the element with the id animation-container and assign it to a variable called animationContainer.

Step 88:

Next, clear out your showAnimation() function by removing all of your setTimeout() calls.

Step 89:

Now you'll start building the animation itself.

First, set the innerText property of result to "Call Stack Animation".

Step 90:

Next, use the .forEach() method to loop through the animationData array. For the .forEach() method's callback function, pass in obj as a parameter, but leave the body of the callback function empty for now.

Step 91:

Since you have the timing for each frame of animation stored in addElDelay, you can use that value with setTimeout() to set up the delay to add elements to the DOM.

Within the body of the .forEach() method's callback function, add a setTimeout() function. Pass in an empty callback function as the first argument, and obj.addElDelay as the second argument.

Step 92:

Then, use the compound assignment operator (+=) to set the innerHTML property of the animationContainer to an empty template literal string.

Step 93:

Within the template literal, add a paragraph element with the `id` attribute equal to an empty string.

Step 94:

Next, use string interpolation to set the `id` attribute to the `inputVal` property of the current object, `obj`.

Step 95:

Add a `class` attribute set to `"animation-frame"`.

Step 96:

Finally, use string interpolation to set the text of the paragraph element to `decimalToBinary(${currVal})`, where `currVal` is the `inputVal` property of the current object. After this, test out your code by entering the number `5` into the number input and clicking the `Convert` button.

Step 97:

Now it's time to set up for the next phase of the animation where you'll update and remove the paragraphs you append to the DOM during the animation.

Add the property `msg` to the animation object at the top of the stack, and set its value to an empty string.

Step 98:

Set the value of the `msg` property to the following string:

Example Code:

"decimalToBinary(1) returns '1' (base case) and gives tha

Step 99:

Next, add the property showMsgDelay with the value 5000 and removeElDelay with the value 10000.

Step 100:

For the object in the middle of the stack, add the property msg set to the following string:

Example Code:

"decimalToBinary(2) returns '1' + 0 (2 % 2) and gives that value to the stack below. Then it pops off the stack."

Also, add the property showMsgDelay set to 10000 and the property removeElDelay set to 15000.

Step 101:

For the last animation object, add the property msg set to the following string:

Example Code:

"decimalToBinary(5) returns '10' + 1 (5 % 2). Then it pops off the stack."

Also, add the property `showMsgDelay` set to `15000` and the property `removeElDelay` set to `20000`.

Step 102:

For the next phase of the animation you'll update the paragraphs with the `msg` text. Since you have the delays for each step of the animation already, you can add your code to the same `.forEach()` loop.

Add another `setTimeout()` function. Pass in an empty callback function as the first argument, and pass in the `showMsgDelay` property of the current object as the second argument.

Step 103:

You have set the `id` attribute for your paragraph elements to the `obj.inputVal` property.

Now, use the `.getElementById()` method to select the element with that attribute value, again using the `obj.inputVal` property.

After that, set the `textContent` property of the selected element equal to the `msg` property of the current object, to update its text after the delay you specified earlier.

Step 104:

Next, you'll remove the paragraph elements from the `#show-animation` element after the delays you specified earlier.

Add a `setTimeout()` function to your `.forEach()` loop. Pass in an empty callback function as the first argument, and pass in the `removeElDelay` property of the current object as the second argument.

Step 105:

Use the .getElementById() method to target the element with the id attribute with the value of the inputVal property of the current object. Then, use the .remove() method on that element to remove it from the DOM after the delay.

Step 106:

Now your animation is complete. When you enter 5 in the number input and click the Convert button, the animation will add paragraphs to the DOM, update the text of each paragraph, and then remove the paragraphs from the DOM.

The last thing you need to do is add the result of converting the number 5 into binary to the page once the animation is complete.

After the .forEach() method, add another setTimeout() function. Pass in an empty callback function as the first argument, and a delay of 20000 milliseconds as the second argument.

Step 107:

Finally, set the textContent property of result equal to calling decimalToBinary() with 5 as an argument. After this, test out your code by entering the number 5 into the number input and clicking the Convert button.

Congratulations! You just finished your decimal to binary converter with recursion.