**Use Recursion to Create a Range of Numbers**

Continuing from the previous challenge, we provide you another opportunity to create a recursive function to solve a problem.

function rangeOfNumbers(startNum, endNum) {

  if (endNum - startNum === 0) {

    return [startNum];

  } else {

    var num = rangeOfNumbers(startNum, endNum - 1);

    num.push(endNum);

    return num;

  }

}

# Compare Scopes of the var and let Keywords

If you are unfamiliar with let, check out [this challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/explore-differences-between-the-var-and-let-keywords).

When you declare a variable with the var keyword, it is declared globally, or locally if declared inside a function.

The let keyword behaves similarly, but with some extra features. When you declare a variable with the let keyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.

function checkScope() {

  "use strict";

  let i = "function scope";

  if (true) {

    let i = "block scope";

    console.log("Block scope i is: ", i);

  }

  console.log("Function scope i is: ", i);

  return i;

}

# Mutate an Array Declared with const

If you are unfamiliar with const, check out [this challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/declare-a-read-only-variable-with-the-const-keyword).

The const declaration has many use cases in modern JavaScript.

Some developers prefer to assign all their variables using const by default, unless they know they will need to reassign the value. Only in that case, they use let.

However, it is important to understand that objects (including arrays and functions) assigned to a variable using const are still mutable. Using the const declaration only prevents reassignment of the variable identifier

const s = [5, 7, 2];

function editInPlace() {

  // Only change code below this line

  "use strict";

  s[0] = 2;

  s[1] = 5;

  s[2] = 7;

}

editInPlace();

# Prevent Object Mutation

As seen in the previous challenge, const declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function Object.freeze to prevent data mutation.

Any attempt at changing the object will be rejected, with an error thrown if the script is running in strict mode.

**function** **freezeObj**() {

**const** MATH\_CONSTANTS = {

PI: 3.14

};

*// Only change code below this line*

Object.freeze(MATH\_CONSTANTS);

*// Only change code above this line*

**try** {

MATH\_CONSTANTS.PI = 99;

} **catch**(ex) {

console.log(ex);

}

**return** MATH\_CONSTANTS.PI;

}

**const** PI = freezeObj();

# Use Arrow Functions to Write Concise Anonymous Functions

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else.

const magic = () => {

  return new Date();

};

# Write Arrow Functions with Parameters

Just like a regular function, you can pass arguments into an arrow function.

const myConcat = (arr1, arr2) => {

  "use strict";

  return arr1.concat(arr2);

};

// test your code

console.log(myConcat([1, 2], [3, 4, 5]));

# Set Default Parameters for Your Functions

In order to help us create more flexible functions, ES6 introduces default parameters for functions.

const increment = (number, value = 1) => number + value;

console.log(increment(5, 2)); // returns 7

console.log(increment(5)); // returns 6

# Use the Rest Parameter with Function Parameters

In order to help us create more flexible functions, ES6 introduces the rest parameter for function parameters. With the rest parameter, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.

const sum = (...args) => {

  return args.reduce((a, b) => a + b, 0);

}

console.log(sum(1, 2, 3)); // 6

# Use the Spread Operator to Evaluate Arrays In-Place

ES6 introduces the spread operator, which allows us to expand arrays and other expressions in places where multiple parameters or elements are expected.

const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];

let arr2

arr2 = [...arr1];  // Change this line

console.log(arr2);

# Use Destructuring Assignment to Extract Values from Objects

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object.

const HIGH\_TEMPERATURES = {

  yesterday: 75,

  today: 77,

  tomorrow: 80

};

const {today, tomorrow} = HIGH\_TEMPERATURES;

# Use Destructuring Assignment to Assign Variables from Objects

Destructuring allows you to assign a new variable name when extracting values. You can do this by putting the new name after a colon when assigning the value.

const HIGH\_TEMPERATURES = {

  yesterday: 75,

  today: 77,

  tomorrow: 80

};

// change code below this line

const { today: highToday, tomorrow: highTomorrow } = HIGH\_TEMPERATURES;

// change code above this line

console.log(highToday);

console.log(highTomorrow);

**Use Destructuring Assignment to Assign Variables from Nested Objects**

const LOCAL\_FORECAST = {

  yesterday: { low: 61, high: 75 },

  today: { low: 64, high: 77 },

  tomorrow: { low: 68, high: 80 }

};

// Only change code below this line

const { today: { low: lowToday, high: highToday } } = LOCAL\_FORECAST;

// Only change code above this line

# Use Destructuring Assignment to Assign Variables from Arrays

ES6 makes destructuring arrays as easy as destructuring objects.

**let** a = 8, b = 6;

[a, b] = [b, a];

**Use Destructuring Assignment with the Rest Parameter to Reassign Array Elements**

const source = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

function removeFirstTwo(list) {

  "use strict";

  // change code below this line

  const [a, b, ...arr] = list;

  // change code above this line

  return arr;

}

const arr = removeFirstTwo(source);

console.log(arr); // should be [3,4,5,6,7,8,9,10]

console.log(source); // should be [1,2,3,4,5,6,7,8,9,10];

**Use Destructuring Assignment to Pass an Object as a Function's Parameters**

const stats = {

  max: 56.78,

  standard\_deviation: 4.34,

  median: 34.54,

  mode: 23.87,

  min: -0.75,

  average: 35.85

};

// Only change code below this line

const half = ({ max, min }) => (max + min) / 2.0;

// Only change code above this line

Create Strings using Template Literals

const result = {

  success: ["max-length", "no-amd", "prefer-arrow-functions"],

  failure: ["no-var", "var-on-top", "linebreak"],

  skipped: ["no-extra-semi", "no-dup-keys"]

};

function makeList(arr) {

  "use strict";

  // change code below this line

  const failureItems = arr.map(item => `<li class="text-warning">${item}</li>`);

  // change code above this line

  return failureItems;

}

const failuresList = makeList(result.failure);

# Write Concise Object Literal Declarations Using Object Property Shorthand

ES6 adds some nice support for easily defining object literals.

const createPerson = (name, age, gender) => {

  "use strict";

  // change code below this line

  return {

    name,

    age,

    gender

  };

  // change code above this line

};

# Write Concise Declarative Functions with ES6

When defining functions within objects in ES5, we have to use the keyword function as follows:

const bicycle = {

  gear: 2,

  setGear(newGear) {

    "use strict";

    this.gear = newGear;

  }

};

# Use class Syntax to Define a Constructor Function

ES6 provides a new syntax to create objects, using the class keyword.

class Vegetable {

  constructor(name) {

    this.name = name;

  }

}

const carrot = new Vegetable("carrot");

console.log(carrot.name); // => should be 'carrot'

# Use getters and setters to Control Access to an Object

You can obtain values from an object and set the value of a property within an object.

class Thermostat {

  constructor(fahrenheit) {

    this.fahrenheit = fahrenheit;

  }

  get temperature() {

    return (5 / 9) \* (this.fahrenheit - 32);

  }

  set temperature(celsius) {

    this.fahrenheit = (celsius \* 9.0) / 5 + 32;

  }

}

# Create a Module Script

JavaScript started with a small role to play on an otherwise mostly HTML web. Today, it’s huge, and some websites are built almost entirely with JavaScript. In order to make JavaScript more modular, clean, and maintainable; ES6 introduced a way to easily share code among JavaScript files.

<html>

  <body>

    <!-- add your code below -->

    <script type="module" src="index.js"></script>

    <!-- add your code above -->

  </body>

</html>

# Use export to Share a Code Block

Imagine a file called math\_functions.js that contains several functions related to mathematical operations. One of them is stored in a variable, add, that takes in two numbers and returns their sum. You want to use this function in several different JavaScript files. In order to share it with these other files, you first need to export it.

export const uppercaseString = (string) => {

  return string.toUpperCase();

}

export const lowercaseString = (string) => {

  return string.toLowerCase()

}

# Reuse JavaScript Code Using import

import allows you to choose which parts of a file or module to load. In the previous lesson, the examples exported add from the math\_functions.js file. Here's how you can import it to use in another file:

import { uppercaseString, lowercaseString } from './string\_functions.js';

// add code above this line

uppercaseString("hello");

lowercaseString("WORLD!");

# Use \* to Import Everything from a File

Suppose you have a file and you wish to import all of its contents into the current file. This can be done with the import \* as syntax. Here's an example where the contents of a file named math\_functions.js are imported into a file in the same directory:

import \* as stringFunctions from "./string\_functions.js";

// add code above this line

stringFunctions.uppercaseString("hello");

stringFunctions.lowercaseString("WORLD!");

# Create an Export Fallback with export default

In the export lesson, you learned about the syntax referred to as a named export. This allowed you to make multiple functions and variables available for use in other files.

"use strict";

export default function subtract(x, y) {

  return x - y;

}

# Import a Default Export

In the last challenge, you learned about export default and its uses. To import a default export, you need to use a different import syntax. In the following example, add is the default export of the math\_functions.js file. Here is how to import it:

import subtract from "./math\_functions.js";

// add code above this line

subtract(7,4);

# Create a JavaScript Promise

A promise in JavaScript is exactly what it sounds like - you use it to make a promise to do something, usually asynchronously. When the task completes, you either fulfill your promise or fail to do so.

const makeServerRequest = new Promise((resolve, reject) => {

});

# Complete a Promise with resolve and reject

A promise has three states: pending, fulfilled, and rejected. The promise you created in the last challenge is forever stuck in the pending state because you did not add a way to complete the promise.

const makeServerRequest = new Promise((resolve, reject) => {

  // responseFromServer represents a response from a server

  let responseFromServer;

  if(responseFromServer) {

    resolve("We got the data");

  } else {

    reject("Data not received");

  }

});

# Handle a Fulfilled Promise with then

Promises are most useful when you have a process that takes an unknown amount of time in your code (i.e. something asynchronous), often a server request.

const makeServerRequest = new Promise((resolve, reject) => {

  // responseFromServer is set to true to represent a successful response from a server

  let responseFromServer = true;

  if(responseFromServer) {

    resolve("We got the data");

  } else {

    reject("Data not received");

  }

});

makeServerRequest.then(result => {

  console.log(result);

});

# Handle a Rejected Promise with catch

catch is the method used when your promise has been rejected. It is executed immediately after a promise's reject method is called. Here’s the syntax:

const makeServerRequest = new Promise((resolve, reject) => {

  // responseFromServer is set to false to represent an unsuccessful response from a server

  let responseFromServer = false;

  if(responseFromServer) {

    resolve("We got the data");

  } else {

    reject("Data not received");

  }

});

makeServerRequest.then(result => {

  console.log(result);

});

makeServerRequest.catch(error => {

  console.log(error);

});

# Using the Test Method

Regular expressions are used in programming languages to match parts of strings. You create patterns to help you do that matching.

let myString = "Hello, World!";

let myRegex = /Hello/;

let result = myRegex.test(myString); // Change this line

# Match Literal Strings

In the last challenge, you searched for the word Hello using the regular expression /Hello/. That regex searched for a literal match of the string Hello. Here's another example searching for a literal match of the string Kevin:

let waldoIsHiding = "Somewhere Waldo is hiding in this text.";

let waldoRegex = /Waldo/; // Change this line

let result = waldoRegex.test(waldoIsHiding);

# Match a Literal String with Different Possibilities

Using regexes like /coding/, you can look for the pattern coding in another string.

let petString = "James has a pet cat.";

let petRegex = /dog|cat|bird|fish/;

let result = petRegex.test(petString);

# Ignore Case While Matching

Up until now, you've looked at regexes to do literal matches of strings. But sometimes, you might want to also match case differences.

let myString = "freeCodeCamp";

let fccRegex = /freeCodeCamp/i;

let result = fccRegex.test(myString);

# Extract Matches

So far, you have only been checking if a pattern exists or not within a string. You can also extract the actual matches you found with the .match() method.

**let** extractStr = "Extract the word 'coding' from this string.";

**let** codingRegex = /coding/;

**let** result = extractStr.match(codingRegex);

# Find More Than the First Match

So far, you have only been able to extract or search a pattern once.

let twinkleStar = "Twinkle, twinkle, little star";

let starRegex = /twinkle/gi;

let result = twinkleStar.match(starRegex);

# Use Recursion to Create a Countdown

In a [previous challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/replace-loops-using-recursion), you learned how to use recursion to replace a for loop. Now, let's look at a more complex function that returns an array of consecutive integers starting with 1 through the number passed to the function.

function countdown(n) {

  if (n < 1) {

    return [];

  } else {

    const arr = countdown(n - 1);

    arr.unshift(n);

    return arr;

  }

}

# Use Recursion to Create a Range of Numbers

Continuing from the previous challenge, we provide you another opportunity to create a recursive function to solve a problem.

function rangeOfNumbers(startNum, endNum) {

  if (endNum - startNum === 0) {

    return [startNum];

  } else {

    var numbers = rangeOfNumbers(startNum, endNum - 1);

    numbers.push(endNum);

    return numbers;

  }

}

# \Compare Scopes of the var and let Keywords

If you are unfamiliar with let, check out [this challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/explore-differences-between-the-var-and-let-keywords).

function checkScope() {

  "use strict";

  let i = "function scope";

  if (true) {

    let i = "block scope";

    console.log("Block scope i is: ", i);

  }

  console.log("Function scope i is: ", i);

  return i;

}

# Return Early Pattern for Functions

When a return statement is reached, the execution of the current function stops and control returns to the calling location.

// Setup

function abTest(a, b) {

  // Only change code below this line

  if (a < 0 || b < 0) {

    return undefined;

  }

  // Only change code above this line

  return Math.round(Math.pow(Math.sqrt(a) + Math.sqrt(b), 2));

}

// Change values below to test your code

abTest(2, 2);

# Counting Cards

In the casino game Blackjack, a player can determine whether they have an advantage on the next hand over the house by keeping track of the relative number of high and low cards remaining in the deck. This is called Card Counting.

let count = 0;

function cc(card) {

  // Only change code below this line

  switch (card) {

    case 2:

    case 3:

    case 4:

    case 5:

    case 6:

      count++;

      break;

    case 10:

    case "J":

    case "Q":

    case "K":

    case "A":

      count--;

      break;

  }

  if (count > 0) {

    return count + " Bet";

  } else {

    return count + " Hold";

  }

  // Only change code above this line

}

cc(2); cc(3); cc(7); cc('K'); cc('A');

# Comparison with the Less Than Operator

The less than operator (<) compares the values of two numbers. If the number to the left is less than the number to the right, it returns true.

function testLessThan(val) {

  if (val < 25) return "Under 25";

  if (val < 55) return "Under 55";

  return "55 or Over";

}

# Comparison with the Less Than Or Equal To Operator

The less than or equal to operator (<=) compares the values of two numbers. If the number to the left is less than or equal to the number to the right, it returns true.

function testLessOrEqual(val) {

  if (val <= 12) return "Smaller Than or Equal to 12";

  if (val <= 24) return "Smaller Than or Equal to 24";

  return "More Than 24";

}