FullStack.Cafe - Kill Your Tech Interview

Q1: Sum of Array Plus One ☆

Topics: JavaScript

Problem:

Write a function that takes an array of integers and returns the sum of the integers after adding 1 to each.

Solution:

```
// ES5 method is nice and clean
exports.es5 = function (array) {
  return array.reduce(function (memo, num) {
    return memo + num;
  }, array.length);
};

// Without array.reduce method isn't much different
exports.iterative = function (array) {
  var result = array.length;

  for (var i = 0; i < array.length; i++) {
    result += array[i];
  }

  return result;
};
```

Q2: String Rotation ☆

Topics: JavaScript

Problem:

Find out if a string is a rotation of another string. E.g. `ABCD` is a rotation of `BCDA` but not `ACBD`.

Solution:

First make sure `a` and `b` are of the same length. Then check to see if `b` is a substring of `a` concatenated with `a`:

```
module.exports = function (a, b) {
  return a.length === b.length && (a + a).indexOf(b) > -1;
};
```

Q3: Oddball sum ☆

Topics: JavaScript

Problem:

Write a function called `oddball_sum` which takes in a list of numbers and returns the sum of all the odd elements. Try to solve with and without `reduce` function.

Solution:

To solve this challenge we'll simply loop through the array while maintaining a final count, and every time an odd number is encountered we'll add it to the count.

Without `reduce`:

```
function oddball_sum(nums) {

  // final count of all odd numbers added up
  var final_count = 0;

  // loop through entire list
  for (var i = 0; i < nums.length; i++) {

    // we divide by 2, and if there is a remainder then
    // the number must be odd so we add it to final_count
    if (nums[i] % 2 === 1) {
      final_count += nums[i]
    }

  }

  return final_count;

}

oddball_sum([1, 2, 3, 4, 5]);
```

With `reduce`:

```
function oddball_sum(nums) {
  return nums.reduce(function(total, item){
      if (item % 2 === 1) {
          return total += item;
      }
      return total;
  });
}
```

Q4: Simple clock angle ☆

Topics: JavaScript

Problem:

You will be given a number `N` that represents where the minute hand currently is on a clock. Your program should return the angle that is formed by the minute hand and the `12` o'clock mark on the clock.

Solution:

If the input is `15` then your program should return `90` because a `90`-degree angle is formed by the minute hand and the `12` o'clock mark on the clock. We'll solve this challenge by first calculating what angle is created by each minute passing on a clock. Once we calculate this number, we multiply it by the input to determine the final angle.

A method to solve such problems is to consider the rate of change of the angle in degrees per minute. The hour hand of a normal `12-hour` analogue clock turns `360°` in `12` hours (`720` minutes) or `0.5°` per minute. The minute hand rotates through `360°` in `60` minutes or `6°` per minute.

```
function simpleClockAngle(num) {

  // we got 6 because 360/60 = 6
  // 360 represents the full number of a degrees in a circle and
  // 60 is the number of minutes on a clock, so dividing these two numbers
  // gives us the number of degrees for one minute
  return 6 * num;

}

simpleClockAngle(15);
```

Q5: Test divisors of three ☆

Topics: JavaScript

Problem:

You will be given 2 parameters: a low and high number. Your goal is to print all numbers between low and high, and for each of these numbers print whether or not the number is divisible by 3. If the number is divisible by 3, print the word "div3" directly after the number.

Solution:

We'll solve this problem by first creating a loop that will print each number from low to high. Once we have the code for that written, we'll add a conditional that will check if the number is evenly divisible by 3 by using the mod operator.

```
function test_divisors(low, high) {

  // we'll store all numbers and strings within an array
  // instead of printing directly to the console
  var output = [];

  for (var i = low; i <= high; i++) {

    // simply store the current number in the output array
    output.push(i);

    // check if the current number is evenly divisible by 3
    if (i % 3 === 0) { output.push('div3'); }

  }

  // return all numbers and strings
  return output;

}

test_divisors(2, 10);
```

Q6: Sum of several arrays ☆

Topics: JavaScript

Problem:

You will be given an array of several arrays that each contain integers and your goal is to write a function that will sum up all the numbers in all the arrays. For example, if the input is `[[3, 2], [1], [4, 12]]` then your program should output `22` because `3 + 2 + 1 + 4 + 12 = 22`. Solve without and with `reduce`.

Solution:

We will solve this challenge by looping through the entire array, and then looping through each inner array adding up all the numbers.

```
function sum_array(arr) {
  // store our final answer
  var sum = 0;
  // loop through entire array
  for (var i = 0; i < arr.length; i++) {
    // loop through each inner array
    for (var j = 0; j < arr[i].length; j++) {
      // add this number to the current final sum
      sum += arr[i][j];
    }
  }

  return sum;
}

sum_array([[3, 2], [1], [4, 12]]);
```

With `reduce`:

```
function sumArray(arr) {
  return arr.reduce((t, e) => t.concat(e)).reduce((t, e) => t + e)
}
```

Q7: Lucky sevens ☆

Topics: JavaScript

Problem:

Write a function called `lucky_sevens` which takes an array of integers and returns true if any three consecutive elements sum to 7.

Solution:

To solve this challenge we'll simply loop through the array starting at the 3rd position, and checking if the number at this index plus the two previous elements sums to 7. We continue doing this as we loop through the entire array. Once we find three elements that sum to 7, we simply return `true`. If we reach the end of the array without finding elements that sum to 7, we return `false`.

```
function lucky_sevens(arr) {

  // if less than 3 elements then this challenge is not possible
  if (arr.length < 3) {
    return "not possible";
  }

  // because we know there are at least 3 elements we can
  // start the loop at the 3rd element in the array (i=2)
  // and check it along with the two previous elements (i-1) and (i-2)
  for (var i = 2; i < arr.length; i++) {
    if (arr[i] + arr[i-1] + arr[i-2] === 7) {
      return true;
    }
  }

  // if loop is finished and no elements summed to 7
  return false;

}

lucky_sevens([2, 1, 5, 1, 0]);
```

Q8: Find the missing number in `O(n)` time ☆☆

Topics: JavaScript

Problem:

Being told that an unsorted array contains `(n - 1)` of `n` consecutive numbers (where the bounds are defined), find the missing number in `O(n)` time

Solution:

```
// The output of the function should be 8
var arrayOfIntegers = [2, 5, 1, 4, 9, 6, 3, 7];
var upperBound = 9;
var lowerBound = 1;

findMissingNumber(arrayOfIntegers, upperBound, lowerBound); // 8

function findMissingNumber(arrayOfIntegers, upperBound, lowerBound) {
  // Iterate through array to find the sum of the numbers
  var sumOfIntegers = 0;
  for (var i = 0; i < arrayOfIntegers.length; i++) {
    sumOfIntegers += arrayOfIntegers[i];
  }

  // Find theoretical sum of the consecutive numbers using a variation of Gauss Sum.
  // Formula: [(N * (N + 1)) / 2] - [(M * (M - 1)) / 2];
  // N is the upper bound and M is the lower bound

  upperLimitSum = (upperBound * (upperBound + 1)) / 2;
  lowerLimitSum = (lowerBound * (lowerBound - 1)) / 2;

  theoreticalSum = upperLimitSum - lowerLimitSum;

  return theoreticalSum - sumOfIntegers;
}
```

Q9: Remove duplicates of an array and return an array of only unique elements ☆☆

Topics: JavaScript

Answer:

```
// ES6 Implementation
var array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8];

Array.from(new Set(array)); // [1, 2, 3, 5, 9, 8]

// ES5 Implementation
var array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8];

uniqueArray(array); // [1, 2, 3, 5, 9, 8]

function uniqueArray(array) {
  var hashmap = {};
  var unique = [];

  for(var i = 0; i < array.length; i++) {
    // If key returns undefined (unique), it is evaluated as false.
    if(!hashmap.hasOwnProperty(array[i])) {
      hashmap[array[i]] = 1;
      unique.push(array[i]);
    }
  }

  return unique;
}
```

Q10: Given a string, reverse each word in the sentence ☆☆

Topics: JavaScript

Problem:

For example `Welcome to this Javascript Guide!` should be become `emocleW ot siht tpircsavaJ !ediuG`

Solution:

```
var string = "Welcome to this Javascript Guide!";

// Output becomes !ediuG tpircsavaJ siht ot emocleW
var reverseEntireSentence = reverseBySeparator(string, "");

// Output becomes emocleW ot siht tpircsavaJ !ediuG
var reverseEachWord = reverseBySeparator(reverseEntireSentence, " ");

function reverseBySeparator(string, separator) {
  return string.split(separator).reverse().join(separator);
}
```

Q11: Implement enqueue and dequeue using only two stacks ☆☆

Topics: JavaScript

Answer:

Enqueue means to add an element, *dequeue* to remove an element.

```
var inputStack = []; // First stack
var outputStack = []; // Second stack

// For enqueue, just push the item into the first stack
function enqueue(stackInput, item) {
  return stackInput.push(item);
}

function dequeue(stackInput, stackOutput) {
  // Reverse the stack such that the first element of the output stack is the
  // last element of the input stack. After that, pop the top of the output to
  // get the first element that was ever pushed into the input stack
  if (stackOutput.length <= 0) {
    while(stackInput.length > 0) {
      var elementToOutput = stackInput.pop();
      stackOutput.push(elementToOutput);
    }
  }

  return stackOutput.pop();
}
```

Q12: Write a "mul" function which will properly when invoked as below syntax ☆☆

Topics: JavaScript

Problem:

```
console.log(mul(2)(3)(4)); // output : 24
console.log(mul(4)(3)(4)); // output : 48
```

Solution:

```
function mul (x) {
  return function (y) { // anonymous function
    return function (z) { // anonymous function
      return x * y * z;
    };
  };
}
```

Here `mul` function accept the first argument and return anonymous function which take the second parameter and return anonymous function which take the third parameter and return multiplication of arguments which is being passed in successive

In JavaScript function defined inside has access to outer function variable and function is the first class object so it can be returned by function as well and passed as argument in another function.

- A function is an instance of the Object type
- A function can have properties and has a link back to its constructor method
- Function can be stored as variable
- Function can be pass as a parameter to another function
- Function can be returned from function

Q13: How to empty an array in JavaScript? ☆☆

Topics: JavaScript

Problem:

```
var arrayList =  ['a', 'b', 'c', 'd', 'e', 'f'];
```

How could we empty the array above?

Solution:

Method 1

```
arrayList = [];
```

Above code will set the variable `arrayList` to a new empty array. This is recommended if you don't have **references to the original array** `arrayList` anywhere else because It will actually create a new empty array. You should be careful with this way of empty the array, because if you have referenced this array from another variable, then the original reference array will remain unchanged, Only use this way if you have only referenced the array by its original variable `arrayList`.

For Instance:

```
var arrayList = ['a', 'b', 'c', 'd', 'e', 'f']; // Created array
var anotherArrayList = arrayList;  // Referenced arrayList by another variable
arrayList = []; // Empty the array
console.log(anotherArrayList); // Output ['a', 'b', 'c', 'd', 'e', 'f']
```

Method 2

```
arrayList.length = 0;
```

Above code will clear the existing array by setting its length to 0. This way of empty the array also update all the reference variable which pointing to the original array. This way of empty the array is useful when you want to update all the another reference variable which pointing to `arrayList`.

For Instance:

```
var arrayList = ['a', 'b', 'c', 'd', 'e', 'f']; // Created array
var anotherArrayList = arrayList;  // Referenced arrayList by another variable
arrayList.length = 0; // Empty the array by setting length to 0
console.log(anotherArrayList); // Output []
```

Method 3

```
arrayList.splice(0, arrayList.length);
```

Above implementation will also work perfectly. This way of empty the array will also update all the references of the original array.

```
var arrayList = ['a', 'b', 'c', 'd', 'e', 'f']; // Created array
var anotherArrayList = arrayList;  // Referenced arrayList by another variable
arrayList.splice(0, arrayList.length); // Empty the array by setting length to 0
console.log(anotherArrayList); // Output []
```

Method 4

```
while(arrayList.length) {
  arrayList.pop();
}
```

Above implementation can also empty the array. But not recommended to use often.

Q14: How to check if an object is an array or not? Provide some code. ☆☆

Topics: JavaScript

Answer:

> The best way to find whether an object is instance of a particular class or not using `toString` method from `Object.prototype`

```
var arrayList = [1 , 2, 3];
```

One of the best use cases of type checking of an object is when we do method overloading in JavaScript. For understanding this let say we have a method called `greet` which take one single string and also a list of string, so making our `greet` method workable in both situation we need to know what kind of parameter is being passed, is it single value or list of value?

```
function greet(param) {
  if() {
    // here have to check whether param is array or not
  }
  else {
  }
}
```

However, in above implementation it might not necessary to check type for array, we can check for single value string and put array logic code in else block, let see below code for the same.

```
 function greet(param) {
   if(typeof param === 'string') {
   }
   else {
     // If param is of type array then this block of code would execute
   }
 }
```

Now it's fine we can go with above two implementations, but when we have a situation like a parameter can be `single value`, `array`, and `object` type then we will be in trouble.

Coming back to checking type of object, As we mentioned that we can use `Object.prototype.toString`

```
if(Object.prototype.toString.call(arrayList) === '[object Array]') {
  console.log('Array!');
}
```

If you are using `jQuery` then you can also used jQuery `isArray` method:

```
if($.isArray(arrayList)) {
  console.log('Array');
} else {
  console.log('Not an array');
}
```

FYI jQuery uses `Object.prototype.toString.call` internally to check whether an object is an array or not.

In modern browser, you can also use:

```
Array.isArray(arrayList);
```

`Array.isArray` is supported by Chrome 5, Firefox 4.0, IE 9, Opera 10.5 and Safari 5

Q15: Two sum problem ☆☆

Topics: JavaScript

Problem:

Given an integer `x` and a sorted array a of `N` distinct integers, design a linear-time algorithm to determine if there exists two distinct indices `i` and `j` such that `a[i] + a[j] == x`

For example, if the array is `[3, 5, 2, -4, 8, 11]` and the sum is `7`, your program should return `[[11, -4], [2, 5]]` because `11 + -4 = 7` and `2 + 5 = 7`.

Solution:

The algorithm below makes use of hash tables which have a constant lookup time. As we pass through each element in the array, we check to see if S minus the current element exists in the hash table. We only need to loop through the array once, resulting in a running time of `O(n)` since each lookup and insertion in a hash table is `O(1)`.

```
// our two sum function which will return
// all pairs in the array that sum up to S
function twoSum(arr, S) {

  var sums = [];
  var hashTable = {};

  // check each element in array
  for (var i = 0; i < arr.length; i++) {

    // calculate S - current element
    var sumMinusElement = S - arr[i];

    // check if this number exists in hash table
    // if so then we found a pair of numbers that sum to S
    if (hashTable[sumMinusElement.toString()] !== undefined) {
      sums.push([arr[i], sumMinusElement]);
    }

    // add the current number to the hash table
    hashTable[arr[i].toString()] = arr[i];

  }

  // return all pairs of integers that sum to S
  return sums;

}

twoSum([3, 5, 2, -4, 8, 11], 7);
```

Q16: Determine overlapping numbers in ranges ☆☆

Topics: JavaScript

Problem:

You will be given an array with `5` numbers. The first 2 numbers represent a range, and the next two numbers represent another range. The final number in the array is `X`. The goal of your program is to determine if both ranges overlap by at least `X` numbers. For example, in the array `[4, 10, 2, 6, 3]` the ranges `4` to `10` and `2` to `6` overlap by at least `3` numbers `(4, 5, 6)`, so your program should return `true`. Solve with and without looping.

If the array is `[10, 20, 4, 14, 4]` then the ranges are:

```
10 11 12 13 14 15 16 17 18 19 20
4 5 6 7 8 9 10 11 12 13 14
```

These ranges overlap by at least `4` numbers, namely: `10, 11, 12, 13, 14` so your program should return `true`.

Solution:

With loop:

```
function OverlappingRanges(arr) {

  // keep a count of how many numbers overlap
  var counter = 0;

  // loop through one of the ranges
  for (var i = arr[0]; i < arr[1]; i++) {

    // check if a number within the first range exists
    // in the second range
    if (i >= arr[2] && i <= arr[3]) {
      counter += 1;
    }

  }

  // check if the numbers that overlap is equal to or greater
  // than the last number in the array
  return (counter >= arr[4]) ? true : false;
}

OverlappingRanges([4, 10, 2, 6, 3]);
```

Without loop:

```
function overlapping(input){
  var nums1 = listOfNums(input[0], input[1]);
  var nums2 = listOfNums(input[2], input[3]);
  var overlappingNum = 0;

  if(nums1[0] >= nums2[0] && nums1[0] <= nums2[1]){
    overlappingNum =  nums2[1] - nums1[0] + 1;
  } else {
    overlappingNum =  nums1[1] - nums2[0] + 1;
  }
  if(overlappingNum >= input[4]){
    return true;
  }
}

function listOfNums(a, b){
  var start = a;
  var end = b;
  if(a > b){
    start = b;
    end = a;
  }

  return [a, b];
}

var a = [4, 10, 2, 6, 3];
overlapping(a)
```

Q17: Write a function that would allow you to do this? ☆☆

Topics: JavaScript

Problem:

```
var addSix = createBase(6);
addSix(10); // returns 16
addSix(21); // returns 27
```

Solution:

You can create a closure to keep the value passed to the function `createBase` even after the inner function is returned. The inner function that is being returned is created within an outer function, making it a closure, and it has access to the variables within the outer function, in this case the variable `baseNumber`.

```
function createBase(baseNumber) {
  return function(N) {
    // we are referencing baseNumber here even though it was declared
    // outside of this function. Closures allow us to do this in JavaScript
    return baseNumber + N;
  }
}

var addSix = createBase(6);
addSix(10);
addSix(21);
```

Q18: How would you check if a number is an integer? ☆☆

Topics: JavaScript

Answer:

A very simply way to check if a number is a decimal or integer is to see if there is a remainder left when you divide by 1.

```
function isInt(num) {
  return num % 1 === 0;
}

console.log(isInt(4)); // true
console.log(isInt(12.2)); // false
console.log(isInt(0.3)); // false
```

Q19: Explain what a *callback* function is and provide a simple example ☆☆

Topics: JavaScript

Answer:

A `callback` function is a function that is passed to another function as an argument and is executed after some operation has been completed. Below is an example of a simple callback function that logs to the console *after* some operations have been completed.

```
function modifyArray(arr, callback) {
  // do something to arr here
  arr.push(100);
  // then execute the callback function that was passed
  callback();
}

var arr = [1, 2, 3, 4, 5];

modifyArray(arr, function() {
  console.log("array has been modified", arr);
});
```

Q20: Make this work ☆☆

Topics: JavaScript

Problem:

```
duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]
```

Solution:

```
function duplicate(arr) {
  return arr.concat(arr);
}

duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]
```

FullStack.Cafe - Kill Your Tech Interview

Q1: Tree Level Order Print ☆☆

Topics: JavaScript

Problem:

Given a binary tree of integers, print it in level order. The output will contain space between the numbers in the same level, and new line between different levels.

Solution:

```
module.exports = function (root) {
  // Doing a breadth first search using recursion.
  (function walkLevel (children) {
    // Create a new queue for the next level.
    var queue = [],
        output;

    // Use the map function to easily join all the nodes together while pushing
    // it's children into the next level queue.
    output = children.map(function (node) {
      // Assuming the node has children stored in an array.
      queue = queue.concat(node.children || []);
      return node.value;
    }).join(' ');

    // Log the output at each level.
    console.log(output);

    // If the queue has values in it, recurse to the next level and walk
    // along it.
    queue.length && walkLevel(queue);
  })([root]);
};
```

Q2: Stock maximum profit ☆☆

Topics: JavaScript

Problem:

You will be given a list of stock prices for a given day and your goal is to return the maximum profit that could have been made by buying a stock at the given price and then selling the stock later on.

For example if the input is:

```
[45, 24, 35, 31, 40, 38, 11]
```

then your program should return 16 because if you bought the stock at $24 and sold it at 40$, a profit of $16 was made and this is the largest profit that could be made. If no profit could have been made, return -1.

Solution:

We'll solve the challenge the following way:

1. Iterate through each number in the list.
2. At the ith index, get the `i+1` index price and check if it is larger than the ith index price.
3. If so, set `buy_price = i` and `sell_price = i+1`. Then calculate the profit: `sell_price - buy_price`.
4. If a stock price is found that is cheaper than the current `buy_price`, set this to be the new buying price and continue from step 2.
5. Otherwise, continue changing only the `sell_price` and keep `buy_price` set.

This algorithm runs in linear time, making only one pass through the array, so the running time in the worst case is `O(n)`.

```
function StockPicker(arr) {

  var max_profit = -1;
  var buy_price = 0;
  var sell_price = 0;

  // this allows our loop to keep iterating the buying
  // price until a cheap stock price is found
  var change_buy_index = true;

  // loop through list of stock prices once
  for (var i = 0; i < arr.length-1; i++) {

    // selling price is the next element in list
    sell_price = arr[i+1];

    // if we have not found a suitable cheap buying price yet
    // we set the buying price equal to the current element
    if (change_buy_index) { buy_price = arr[i]; }

    // if the selling price is less than the buying price
    // we know we cannot make a profit so we continue to the
    // next element in the list which will be the new buying price
    if (sell_price < buy_price) {
      change_buy_index = true;
      continue;
    }

    // if the selling price is greater than the buying price
    // we check to see if these two indices give us a better
    // profit then what we currently have
    else {
      var temp_profit = sell_price - buy_price;
      if (temp_profit > max_profit) { max_profit = temp_profit; }
      change_buy_index = false;
    }

  }

  return max_profit;

}

StockPicker([44, 30, 24, 32, 35, 30, 40, 38, 15]);
```

Q3: Step-by-step solution for step counting using recursion ☆☆

Topics: JavaScript

Problem:

Suppose you want climb a staircase of N steps, and on each step you can take either 1 or 2 steps. How many distinct ways are there to climb the staircase? For example, if you wanted to climb 4 steps, you can take the following distinct number of steps:

```
1) 1, 1, 1, 1
2) 1, 1, 2
3) 1, 2, 1
4) 2, 1, 1
5) 2, 2
```

So there are 5 distinct ways to climb 4 steps. We want to write a function, using recursion, that will produce the answer for any number of steps.

Solution:

The solution to this problem requires recursion, which means to solve for a particular `N`, we need the solutions for previous N's. The solution for N steps is equal to the solutions for `N - 1` steps plus `N - 2` steps.

```
function countSteps(N) {

  // just as in our solution explanation above, we know that to climb 1 step
  // there is only 1 solution, and for 2 steps there are 2 solutions
  if (N === 1) { return 1; }
  if (N === 2) { return 2; }

  // for all N > 2, we add the previous (N - 1) + (N - 2) steps to get
  // an answer recursively
  return countSteps(N - 1) + countSteps(N - 2);

}

// the solution for N = 6 will recursively be solved by calculating
// the solution for N = 5, N = 4, N = 3, and N = 2 which we know is 2
countSteps(6);
```

Q4: Implement Bubble Sort ☆☆

Topics: JavaScript

Answer:

The steps in the bubble sort algorithm are:

1. Loop through the whole array starting from index `1`
2. If the number in the array at index `i-1` is greater than i, swap the numbers and continue
3. Once the end of the array is reached, start looping again from the beginning
4. Once no more elements can be swapped, the sorting is complete

```
function swap(arr, i1, i2) {
  var temp = arr[i1];
  arr[i1] = arr[i2];
  arr[i2] = temp;
}

function bubblesort(arr) {

  var swapped = true;

  // keep going unless no elements can be swapped anymore
  while (swapped) {

    // set swapped to false so that the loop stops
    // unless two element are actually swapped
    swapped = false;

    // loop through the whole array swapping adjacent elements
    for (var i = 1; i < arr.length; i++) {
      if (arr[i-1] > arr[i]) {
        swap(arr, i-1, i);
        swapped = true;
      }
    }

  }

  return arr;

}

bubblesort([103, 45, 2, 1, 97, -4, 67]);
```

Q5: Provide some examples of non-bulean value coercion to a boolean one ☆☆☆

Topics: JavaScript

Answer:

The question is when a non-boolean value is coerced to a boolean, does it become `true` or `false`, respectively?

The specific list of "falsy" values in JavaScript is as follows:

- `""` (empty string)
- `0`, `-0`, `NaN` (invalid number)
- `null`, `undefined`
- `false`

Any value that's not on this "falsy" list is "truthy." Here are some examples of those:

- `"hello"`
- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` (arrays)
- `{ }`, `{ a: 42 }` (objects)
- `function foo() { .. }` (functions)

Q6: Given an array of integers, find the largest product yielded from three of the integers ☆☆☆

Topics: JavaScript

Answer:

```
var unsortedArray = [-10, 7, 29, 30, 5, -10, -70];

computeProduct(unsortedArray); // 21000
```

```
function sortIntegers(a, b) {
   return a - b;
}

// Greatest product is either (min1 * min2 * max1 || max1 * max2 * max3)
function computeProduct(unsorted) {
   var sortedArray = unsorted.sort(sortIntegers),
       product1 = 1,
       product2 = 1,
       array_n_element = sortedArray.length - 1;

   // Get the product of three largest integers in sorted array
   for (var x = array_n_element; x > array_n_element - 3; x--) {
       product1 = product1 * sortedArray[x];
   }

   product2 = sortedArray[0] * sortedArray[1] * sortedArray[array_n_element];

   if (product1 > product2) return product1;

   return product2;
}
```

Q7: Given an array of integers, find the largest difference between two elements such that the element of lesser value must come before the greater element ☆☆☆

Topics: JavaScript

Answer:

```
var array = [7, 8, 4, 9, 9, 15, 3, 1, 10];
// [7, 8, 4, 9, 9, 15, 3, 1, 10] would return `11` based on the difference between `4` and `15`
// Notice: It is not `14` from the difference between `15` and `1` because 15 comes before 1.

findLargestDifference(array);

function findLargestDifference(array) {
  // If there is only one element, there is no difference
  if (array.length <= 1) return -1;

  // currentMin will keep track of the current lowest
  var currentMin = array[0];
  var currentMaxDifference = 0;

  // We will iterate through the array and keep track of the current max difference
  // If we find a greater max difference, we will set the current max difference to that variable
  // Keep track of the current min as we iterate through the array, since we know the greatest
  // difference is yield from `largest value in future` - `smallest value before it`

  for (var i = 1; i < array.length; i++) {
    if (array[i] > currentMin && (array[i] - currentMin > currentMaxDifference)) {
      currentMaxDifference = array[i] - currentMin;
    } else if (array[i] <= currentMin) {
      currentMin = array[i];
    }
  }

  // If negative or 0, there is no largest difference
  if (currentMaxDifference <= 0) return -1;

  return currentMaxDifference;
}
```

Q8: Find the intersection of two arrays ☆☆☆

Topics: JavaScript

Problem:

An intersection would be the common elements that exists within both arrays. In this case, these elements should be unique!

Solution:

```
var firstArray = [2, 2, 4, 1];
var secondArray = [1, 2, 0, 2];

intersection(firstArray, secondArray); // [2, 1]

function intersection(firstArray, secondArray) {
  // The logic here is to create a hashmap with the elements of the firstArray as the keys.
  // After that, you can use the hashmap's O(1) look up time to check if the element exists in the hash
  // If it does exist, add that element to the new array.

  var hashmap = {};
  var intersectionArray = [];

  firstArray.forEach(function(element) {
    hashmap[element] = 1;
  });

  // Since we only want to push unique elements in our case... we can implement a counter to keep track
of what we already added
  secondArray.forEach(function(element) {
    if (hashmap[element] === 1) {
      intersectionArray.push(element);
      hashmap[element]++;
    }
  });

  return intersectionArray;

  // Time complexity O(n), Space complexity O(n)
}
```

Q9: Given two strings, return true if they are anagrams of one another ☆☆☆

Topics: JavaScript

Problem:

For example: `Mary` is an anagram of `Army`

Solution:

```
var firstWord = "Mary";
var secondWord = "Army";

isAnagram(firstWord, secondWord); // true

function isAnagram(first, second) {
  // For case insensitivity, change both words to lowercase.
  var a = first.toLowerCase();
  var b = second.toLowerCase();
  // Sort the strings, and join the resulting array to a string. Compare the results
  a = a.split("").sort().join("");
  b = b.split("").sort().join("");

  return a === b;
}
```

Q10: Check if a given string is a palindrome. Case sensitivity should be taken into account. ☆☆☆

Topics: JavaScript

Answer:

A **palindrome** is a word, phrase, number, or other sequence of characters which reads the same backward or forward.

```
isPalindrome("racecar"); // true
isPalindrome("race Car"); // true

function isPalindrome(word) {
  // Replace all non-letter chars with "" and change to lowercase
  var lettersOnly = word.toLowerCase().replace(/\s/g, "");

  // Compare the string with the reversed version of the string
  return lettersOnly === lettersOnly.split("").reverse().join("");
}
```

Or `25x` faster than the standard answer

```
function isPalindrome(s,i) {
   return (i=i||0)<0||i>=s.length>>1||s[i]==s[s.length-1-i]&&isPalindrome(s,++i);
}
```

Q11: Write a recursive function that returns the binary string of a given decimal number ☆☆☆

Topics: JavaScript

Answer:

Given 4 as the decimal input, the function should return 100.

```
decimalToBinary(3); // 11
decimalToBinary(8); // 1000
decimalToBinary(1000); // 1111101000

function decimalToBinary(digit) {
  if(digit >= 1) {
    // If digit is not divisible by 2 then recursively return proceeding
    // binary of the digit minus 1, 1 is added for the leftover 1 digit
    if (digit % 2) {
      return decimalToBinary((digit - 1) / 2) + 1;
    } else {
      // Recursively return proceeding binary digits
      return decimalToBinary(digit / 2) + 0;
    }
  } else {
    // Exit condition
    return '';
  }
}
```

Q12: What will be the output of the following code? ☆☆☆

Topics: JavaScript

Problem:

```
var y = 1;
if (function f() {}) {
  y += typeof f;
}
console.log(y);
```

Solution:

Above code would give output `1undefined`. If condition statement evaluate using `eval` so `eval(function f() {})` which return `function f() {}` which is true so inside if statement code execute. `typeof f` return undefined because if statement code execute at run time, so statement inside `if` condition evaluated at run time.

```
var k = 1;
if (1) {
  eval(function foo() {});
  k += typeof foo;
}
console.log(k);
```

Above code will also output `1undefined`.

```
var k = 1;
if (1) {
  function foo() {};
  k += typeof foo;
}
console.log(k); // output 1function
```

Q13: All Permutations (Anagrams) of a String ☆☆☆

Topics: JavaScript

Problem:

Generate all permutations of a given string. (Note: also known as the generating anagrams problem).

Solution:

Remove the first character and recurse to get all permutations of length `N-1`, then insert that first character into `N-1` length strings and obtain all permutations of length `N`. The complexity is `O(N!)` because there are `N!`

possible permutations of a string with length `N`, so it's optimal.

```
module.exports = function (string) {
  var result = {};

  // Using an immediately invoked named function for recursion.
  (function makeWord (word, remaining) {
    // If there are no more remaining characters, break and set to true
    // in the result object.
    if (!remaining) { return result[word] = true; }

    // Loop through all the remaining letters and recurse slicing the character
    // out of the remaining stack and into the solution word.
    for (var i = 0; i < remaining.length; i++) {
      makeWord(
        word + remaining[i],
        remaining.substr(0, i) + remaining.substr(i + 1)
      );
    }
  })('', string);

  // Using the ES5 Object.keys to grab the all the keys as an array.
  return Object.keys(result);
};
```

Q14: Generate all balanced bracket combinations ☆☆☆

Topics: JavaScript

Problem:

Print all possible balanced parenthesis combinations up to `N`. For example:

```
N = 2
(()), ()()

N = 3
((())), (()()), (())(), ()(()), ()()()
```

Solution:

We will implement a recursive function to solve this challenge. The idea is:

1. Add a left bracket to a newly created string.
2. If a left bracket was added, potentially add a new left bracket and add a right bracket.
3. After each of these steps we add the string to an array that stores all bracket combinations.

```
var all = [];

function parens(left, right, str) {

  // if no more brackets can be added then add the final balanced string
  if (left === 0 && right === 0) {
    all.push(str);
  }

  // if we have a left bracket left we add it
  if (left > 0) {
    parens(left-1, right+1, str+"(");
  }
  // if we have a right bracket left we add it
  if (right > 0) {
    parens(left, right-1, str+")");
  }

}

// the parameters parens(x, y, z) specify:
// x: left brackets to start adding
// y: right brackets we can add only after adding a left bracket
// z: the string so far
parens(3, 0, "");
console.log(all);
```

Q15: How would you use a closure to create a private counter? ☆☆☆

Topics: JavaScript

Answer:

You can create a function within an outer function (a closure) that allows you to update a private variable but the variable wouldn't be accessible from outside the function without the use of a helper function.

```
function counter() {
  var _counter = 0;
  // return an object with several functions that allow you
  // to modify the private _counter variable
  return {
    add: function(increment) { _counter += increment; },
    retrieve: function() { return 'The counter is currently at: ' + _counter; }
  }
}

// error if we try to access the private variable like below
// _counter;

// usage of our counter function
var c = counter();
c.add(5);
c.add(9);

// now we can access the private variable in the following way
c.retrieve(); // => The counter is currently at: 14
```

Q16: Implement a queue using two stacks ☆☆☆

Topics: JavaScript

Answer:

Suppose we push `a`, `b`, `c` to a stack. If we are trying to implement a queue and we call the dequeue method 3 times, we actually want the elements to come out in the order: `a`, `b`, `c`, which is in the opposite order they would come out if we popped from the stack. So, basically, we need to access the elements in the reverse order that they exist in the stack.

Algorithm for queue using two stacks:

1. When calling the enqueue method, simply push the elements into the stack 1.

2. If the dequeue method is called, push all the elements from stack 1 into stack 2, which reverses the order of the elements. Now pop from stack 2.

The worst case running time for implementing these operations using stacks is `O(n)` because you need to transfer n elements from stack 1 to stack 2 when a dequeue method is called. Pushing to stack 1 is simply `O(1)`.

```
// implement stacks using plain arrays with push and pop functions
var Stack1 = [];
var Stack2 = [];

// implement enqueue method by using only stacks
// and the push and pop functions
function Enqueue(element) {
  Stack1.push(element);
}

// implement dequeue method by pushing all elements
// from stack 1 into stack 2, which reverses the order
// and then popping from stack 2
function Dequeue() {
  if (Stack2.length === 0) {
    if (Stack1.length === 0) { return 'Cannot dequeue because queue is empty'; }
    while (Stack1.length > 0) {
      var p = Stack1.pop();
      Stack2.push(p);
    }
  }
  return Stack2.pop();
}

Enqueue('a');
Enqueue('b');
Enqueue('c');
Dequeue();
```

Q17: Find all string combinations consisting only of 0, 1 and ? ☆☆☆

Topics: JavaScript

Problem:

The input will be a string consisting only of the characters `0`, `1` and `?`, where the `?` acts as a wildcard that can be either a `0` or `1`, and your goal is to print all possible combinations of the string. For example, if the string is `"011?0"` then your program should output a set of all strings, which are: `["01100", "01110"]`.

Solution:

The general algorithm we will write a solution for is:

1. Call the function with the string and an empty set where we begin pushing `0` and `1`'s.
2. Once we reach a `?` make a copy of each string set, and for half append a `0` and for the other half append a `1`.
3. Recursively call the function with a smaller string until the string is empty.

```
function patterns(str, all) {

  // if the string is empty, return all the string sets
  if (str.length === 0) { return all; }

  // if character is 0 or 1 then add the character to each
  // string set we currently have so far
  if (str[0] === '0' || str[0] === '1') {
    for (var i = 0; i < all.length; i++) {
      all[i].push(str[0]);
    }
  }

  // for a wildcard, we make a copy of each string set
  // and for half of them we append a 0 to the string
  // and for the other half we append a 1 to the string
  if (str[0] === '?') {
    var len = all.length;
    for (var i = 0; i < len; i++) {
      var temp = all[i].slice(0);
      all.push(temp);
    }
    for (var i = 0; i < all.length; i++) {
      (i < all.length/2) ? all[i].push('0') : all[i].push('1');
    }
  }

  // recursively calculate all string sets
  return patterns(str.substring(1), all);

}

patterns('10?1?', [[]]);
```

Q18: What will the following code output? ☆☆☆

Topics: JavaScript

Problem:

```
(function() {
  var a = b = 5;
})();

console.log(b);
```

Solution:

The code above will output 5 even though it seems as if the variable was declared within a function and can't be accessed outside of it. This is because

```
var a = b = 5;
```

is interpreted the following way:

```
var a = b;
b = 5;
```

But `b` is not declared anywhere in the function with `var` so it is set equal to 5 in the *global scope*.

Q19: Write a function that would allow you to do this ☆☆☆

Topics: JavaScript

Problem:

```
multiply(5)(6);
```

Solution:

You can create a *closure* to keep the value of a even after the inner function is returned. The inner function that is being returned is created within an outer function, making it a closure, and it has access to the variables within the outer function, in this case the variable `a`.

```
function multiply(a) {
  return function(b) {
    return a * b;
  }
}

multiply(5)(6);
```

Q20: FizzBuzz Challenge ☆☆☆

Topics: JavaScript

Problem:

Create a for loop that iterates up to `100` while outputting **"fizz"** at multiples of `3`, **"buzz"** at multiples of `5` and **"fizzbuzz"** at multiples of `3` and `5`.

Solution:

Check out this version of FizzBuzz:

```
for (let i = 1; i <= 100; i++) {
  let f = i % 3 == 0,
    b = i % 5 == 0;
  console.log(f ? (b ? 'FizzBuzz' : 'Fizz') : b ? 'Buzz' : i);
}
```

FullStack.Cafe - Kill Your Tech Interview

Q1: Find Word Positions in Text ☆☆☆

Topics: JavaScript

Problem:

Given a text file and a word, find the positions that the word occurs in the file. We'll be asked to find the positions of many words in the same file.

Solution:

Since we'll have to answer multiple queries, precomputation would be useful. We'll build a data structure that stores the positions of all the words in the text file. This is known as inverted index.

```
module.exports = function (text) {
  var trie   = {},
      pos    = 0,
      active = trie; // Start the active structure as the root trie structure

  // Suffix a space after the text to make life easier
  text += ' ';

  // Loop through the input text adding it to the trie structure
  for (var i = 0; i < text.length; i++) {
    // When the character is a space, restart
    if (text[i] === ' ') {
      // If the current active doesn't equal the root, set the position
      if (active !== trie) {
        (active.positions = active.positions || []).push(pos);
      }
      // Reset the positions and the active part of the data structure
      pos    = i;
      active = trie;
      continue;
    }

    // Set the next character in the structure up
    active[text[i]] = (active[text[i]] || {});
    active = active[text[i]];
  }

  // Return a function that accepts a word and looks it up in the trie structure
  return function (word) {
    var i      = -1,
        active = trie;

    while (word[++i]) {
      if (!active[word[i]]) { return []; }
      active = active[word[i]];
    }

    return active.positions;
  };
};
```

Q2: Throttle Function Implementation ☆☆☆

Topics: JavaScript

Problem:

Write a function that accepts a function and timeout, `x`, in number of milliseconds. It returns a function that can only be executed once per `x` milliseconds. This can be useful for limiting the number of time and computation heavy function that are run. For example, making AJAX requests to an autocompletion API.

Once written, add a third parameter that will allow the function to be executed immediately if set to true. Otherwise the function will run at the end of the timeout period.

Solution:

```
module.exports = function (fn, delay, execAsap) {
  var timeout; // Keeps a reference to the timeout inside the returned function

  return function () {
    // Continue to pass through the function execution context and arguments
    var that = this,
        args = arguments;

    // If there is no timeout variable set, proceed to create a new timeout
    if (!timeout) {
      execAsap && fn.apply(that, args);

      timeout = setTimeout(function () {
        execAsap || fn.apply(that, args);
        // Remove the old timeout variable so the function can run again
        timeout = null;
      }, delay || 100);
    }
  };
};
```

Q3: Implement `pow(a,b)` without multiplication or division ☆☆☆

Topics: JavaScript

Problem:

In this challenge we need to implement exponentiation, or raising a to some power of `b` which is usually written `pow(a, b)`. In this variation of the challenge, we also need to implement a solution without using the multiplication or division operations, only addition and subtraction are allowed. Try to solve with recursion and closures.

Solution:

The algorithm to implement:

1. Create a variable named answer
2. Loop from `1` to `n`
3. Each time through the loop we add `a` + `a` + ... (we add a to itself a times) and store result in answer
4. Then each time through the loop we perform step 3 replacing a with answer.

```
// our modified pow function that raises a to the power of b
// without using multiplication or division
function modPow(a, n) {
  // convert a to positive number
  var answer = Math.abs(a);

  // store exponent for later use
  var exp = n;

  // loop n times
  while (n > 1) {

    // add the previous added number n times
    // e.g. 4^3 = 4 * 4 * 4
    //      4*4 = 4 + 4 + 4 + 4 = 16
    //     16*4 = 16 + 16 + 16 + 16 = 64
    var added = 0;
    for (var i = 0; i < Math.abs(a); i++) { added += answer; }
    answer = added;
    n--;

  }

  // if a was negative determine if the answer will be
  // positive or negative based on the original exponent
  // e.g. pow(-4, 3) = (-4)^3 = -64
  return (a < 0 && exp % 2 === 1) ? -answer : answer;

}

modPow(2, 10);
//modPow(5, 4);
//modPow(-4, 7);
```

Using recursion and closures:

```
function pow(num, e) {
  let exponent;
  let value = num;
  addup();

  function addup(outernum = num, iterate = 1) {
    if (iterate == e) {
      return;
    }
    for (let counter = 1; counter < num; counter++) {
      value += outernum;
    }
    exponent = value;
    return addup(value, iterate + 1);
  }
  return exponent;
}
```

Q4: Implement a queue using a linked list ☆☆☆

Topics: JavaScript

Answer:

We will store a reference to the front and back of the queue in order to make enqueuing and dequeuing run in `O(1)` constant time. Every time we want to insert into the queue, we add the new element to the end of the linked list and update the back pointer. When we want to dequeue we return the first node in the linked list and update the front pointer.

```
// queue is initially empty
var Queue = {front: null, back: null};
```

```
// we will use a node to keep track of the elements
// in the queue which is represented by a linked list
function Node(data, next) {
  this.data = data;
  this.next = next;
}

// add elements to queue in O(1) time
function Enqueue(element) {
  var N = new Node(element, null);
  if (Queue.back === null) {
    Queue.front = N;
    Queue.back = N;
  } else {
    Queue.back.next = N;
    Queue.back = Queue.back.next;
  }
}

// remove first element from queue in O(1) time
function Dequeue() {
  if (Queue.front !== null) {
    var first = Queue.front;
    Queue.front = Queue.front.next;
    return first.data;
  } else {
    if (Queue.back !== null) { Queue.back = null; }
    return 'Cannot dequeue because queue is empty';
  }
}

Enqueue('a');
Enqueue('b');
Enqueue('c');
Dequeue();
```

Q5: Merge two sorted linked lists ☆☆☆

Topics: JavaScript

Problem:

The goal here is to merge two linked lists that are already sorted into a new sorted array. For example:

```
L1 = 1 -> 3 -> 10
L2 = 5 -> 6 -> 9
merge(L1, L2) = 1 -> 3 -> 5 -> 6 -> 9 -> 10
```

Solution:

Algorithm:

1. Create a new head pointer to an empty linked list.
2. Check the first value of both linked lists.
3. Whichever node from `L1` or `L2` is smaller, append it to the new list and move the pointer to the next node.
4. Continue this process until you reach the end of a linked list.

```
function Node(data, next) {
  this.data = data;
  this.next = next;
}
```

```
function merge(L1, L2) {

  // create new linked list pointer
  var L3 = new Node(null, null);
  var prev = L3;

  // while both linked lists are not empty
  while (L1 !== null && L2 !== null) {
    if (L1.data <= L2.data) {
      prev.next = L1;
      L1 = L1.next;
    } else {
      prev.next = L2;
      L2 = L2.next;
    }
    prev = prev.next;
  }

  // once we reach end of a linked list, append the other
  // list because we know it is already sorted
  if (L1 === null) { prev.next = L2; }
  if (L2 === null) { prev.next = L1; }

  // return the sorted linked list
  return L3.next;

}

// create first linked list: 1 -> 3 -> 10
var n3 = new Node(10, null);
var n2 = new Node(3, n3);
var n1 = new Node(1, n2);
var L1 = n1;

// create second linked list: 5 -> 6 -> 9
var n6 = new Node(9, null);
var n5 = new Node(6, n6);
var n4 = new Node(5, n5);
var L2 = n4;

merge(L1, L2);
```

Q6: Dutch national flag sorting problem ☆☆☆

Topics: JavaScript

Problem:

For this problem, your goal is to sort an array of `0`, `1`, `2` but you must do this in place, in linear time and without any extra space (such as creating an extra array). This is called the *Dutch national flag sorting problem*. For example, if the input array is `[2,0,0,1,2,1]` then your program should output `[0,0,1,1,2,2]` and the algorithm should run in `O(n)` time.

Solution:

The solution to this algorithm will require 3 pointers to iterate throughout the array, swapping the necessary elements.

1. Create a low pointer at the beginning of the array and a high pointer at the end of the array.
2. Create a mid pointer that starts at the beginning of the array and iterates through each element.
3. If the element at `arr[mid]` is a `2`, then swap `arr[mid]` and `arr[high]` and decrease the high pointer by `1`.
4. If the element at `arr[mid]` is a `0`, then swap `arr[mid]` and `arr[low]` and increase the low and mid pointers by `1`.

5. If the element at `arr[mid]` is a `1`, don't swap anything and just increase the mid pointer by `1`.

```
function swap(arr, i1, i2) {
  var temp = arr[i1];
  arr[i1] = arr[i2];
  arr[i2] = temp;
}

function dutchNatFlag(arr) {

  var low = 0;
  var mid = 0;
  var high = arr.length - 1;

  // one pass through the array swapping
  // the necessary elements in place
  while (mid <= high) {
    if      (arr[mid] === 0) { swap(arr, low++, mid++); }
    else if (arr[mid] === 2) { swap(arr, mid, high--); }
    else if (arr[mid] === 1) { mid++; }
  }

  return arr;

}

dutchNatFlag([2,2,2,0,0,0,1,1]);
```

Q7: Insert an interval into a list of sorted disjoint intervals ☆☆☆

Topics: JavaScript

Problem:

The input is a sorted list of disjoint intervals, and your goal is to insert a new interval and merge all necessary intervals returning a final new list. For example,

```
// if the interval list is
[[1,5], [10,15], [20,25]]
// and you need to insert the interval
[12,27]
// then your program should return the new list:
[[1,5], [10,27]]
```

Solution:

Algorithm:

1. Create an array where the final intervals will be stored.
2. Push all the intervals into this array that come before the new interval you are adding.
3. Once we reach an interval in that comes after the new interval, add our new interval to the final array.
4. From this point, check each remaining element in the array and determine if the intervals need to be merged.

```
function insertInterval(arr, interval) {

  var newSet = [];
  var endSet = [];
  var i = 0;
  // add intervals that come before the new interval
  while (i < arr.length && arr[i][1] < interval[0]) {
    newSet.push(arr[i]);
    i++;
  }

  // add our new interval to this final list
  newSet.push(interval);

  // check each interval that comes after the new interval to determine if we can merge
  // if no merges are required then populate a list of the remaining intervals
  while (i < arr.length) {
    var last = newSet[newSet.length - 1];
    if (arr[i][0] < last[1]) {
      var newInterval = [Math.min.apply(null, [last[0], arr[i][0]]), Math.max.apply(null, [last[1],
arr[i][1]])];
      newSet[newSet.length - 1] = newInterval;
    } else {
      endSet.push(arr[i]);
    }
    i++;
  }

  return newSet.concat(endSet);

}

insertInterval([[1,5],[10,15],[20,25]], [12,27]);
```

Q8: Quickly calculate the cube root of 6 digit numbers ☆☆☆

Topics: JavaScript

Problem:

For example, if the input is `636056` then your program should output `86`.

Solution:

The general algorithm is as follows:

1. Store the first 10 cube roots, their cubes, and the last digit in the number.

```
var cubes_10 = {
    '0': 0,
    '1': 1,
    '8': 8,
   '27': 7,
   '64': 4,
  '125': 5,
  '216': 6,
  '343': 3,
  '512': 2,
  '729': 9
};
```

2. Ignore the last 3 digits of the input number, and for the remaining numbers, find the cube in the table that is less than or equal to the remaining number, and take the corresponding cube root to be the first number in your answer.

3. For the last 3 digits that you previously ignored, loop through the table and when you get to the ith index, where i equals the last digit of the remaining 3 numbers, take the corresponding number in the right column

as your answer.

4. These numbers combined are the cube root answer.

```
function fastCubeRoot(num) {

  var cubes_10 = {
      '0': 0,
      '1': 1,
      '8': 8,
     '27': 7,
     '64': 4,
    '125': 5,
    '216': 6,
    '343': 3,
    '512': 2,
    '729': 9
  };

  // get last 3 numbers and the remaining numbers
  var arr = num.toString().split('');
  var last = arr.slice(-3);
  var first = parseInt(arr.slice(0, -3).join(''));

  // answer will be stored here
  var lastDigit = 0, firstDigit = 0, index = 0;

  // get last digit of cube root
  for (var i in cubes_10) {
    if (index === parseInt(last[last.length-1])) { lastDigit = cubes_10[i]; }
    index++;
  }

  // get first digit of cube root
  index = 0;
  for (var i in cubes_10) {
    if (parseInt(i) <= first) { firstDigit = index; }
    index++;
  }

  // return cube root answer
  return firstDigit + '' + lastDigit;

}

fastCubeRoot(830584);
```

Q9: How does the `this` keyword work? Provide some code examples ☆☆☆☆

Topics: JavaScript

Answer:

In JavaScript *this* always refers to the “owner” of the function we're executing, or rather, to the object that a function is a method of.

Consider:

```
function foo() {
    console.log( this.bar );
}

var bar = "global";
var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

foo();              // "global"
obj1.foo();         // "obj1"
foo.call( obj2 );   // "obj2"
new foo();          // undefined
```

Q10: Create a function that will evaluate if a given expression has balanced parentheses using stacks ☆☆☆☆

Topics: JavaScript

Answer:

In this example, we will only consider `{}` as valid parentheses `{}{}` would be considered balancing. `{{{}}` is not balanced.

```
var expression = "{{}}{}{}"
var expressionFalse = "{}{{}";

isBalanced(expression); // true
isBalanced(expressionFalse); // false
isBalanced(""); // true

function isBalanced(expression) {
  var checkString = expression;
  var stack = [];

  // If empty, parentheses are technically balanced
  if (checkString.length <= 0) return true;

  for (var i = 0; i < checkString.length; i++) {
    if(checkString[i] === '{') {
      stack.push(checkString[i]);
    } else if (checkString[i] === '}') {
      // Pop on an empty array is undefined
      if (stack.length > 0) {
        stack.pop();
      } else {
        return false;
      }
    }
  }

  // If the array is not empty, it is not balanced
  if (stack.pop()) return false;
  return true;
}
```

Q11: Write a recursive function that performs a binary search ☆☆☆☆

Topics: JavaScript

Answer:

```
function recursiveBinarySearch(array, value, leftPosition, rightPosition) {
  // Value DNE
  if (leftPosition > rightPosition) return -1;

  var middlePivot = Math.floor((leftPosition + rightPosition) / 2);
  if (array[middlePivot] === value) {
    return middlePivot;
  } else if (array[middlePivot] > value) {
    return recursiveBinarySearch(array, value, leftPosition, middlePivot - 1);
  } else {
    return recursiveBinarySearch(array, value, middlePivot + 1, rightPosition);
  }
}
```

Q12: Given an integer, determine if it is a power of 2. If so, return that number, else return -1 ☆☆☆☆

Topics: JavaScript

Answer:

Note, 0 is not a power of two.

```
isPowerOfTwo(4); // true
isPowerOfTwo(64); // true
isPowerOfTwo(1); // true
isPowerOfTwo(0); // false
isPowerOfTwo(-1); // false

// For the non-zero case:
function isPowerOfTwo(number) {
  // `&` uses the bitwise n.
  // In the case of number = 4; the expression would be identical to:
  // `return (4 & 3 === 0)`
  // In bitwise, 4 is 100, and 3 is 011. Using &, if two values at the same
  // spot is 1, then result is 1, else 0. In this case, it would return 000,
  // and thus, 4 satisfies are expression.
  // In turn, if the expression is `return (5 & 4 === 0)`, it would be false
  // since it returns 101 & 100 = 100 (NOT === 0)

  return number & (number - 1) === 0;
}

// For zero-case:
function isPowerOfTwoZeroCase(number) {
  return (number !== 0) && ((number & (number - 1)) === 0);
}
```

Q13: What is *Closure* in JavaScript? Provide an example ☆☆☆☆

Topics: JavaScript

Answer:

A *closure* is a function defined inside another function (called parent function) and has access to the variable which is declared and defined in parent function scope.

The closure has access to variable in three scopes:

- Variable declared in his own scope
- Variable declared in parent function scope
- Variable declared in global namespace

```
var globalVar = "abc";

// Parent self invoking function
(function outerFunction (outerArg) { // begin of scope outerFunction
  // Variable declared in outerFunction function scope
  var outerFuncVar = 'x';
  // Closure self-invoking function
  (function innerFunction (innerArg) { // begin of scope innerFunction
    // variable declared in innerFunction function scope
    var innerFuncVar = "y";
    console.log(
      "outerArg = " + outerArg + "\n" +
      "outerFuncVar = " + outerFuncVar + "\n" +
      "innerArg = " + innerArg + "\n" +
      "innerFuncVar = " + innerFuncVar + "\n" +
      "globalVar = " + globalVar);
  // end of scope innerFunction
  })(5); // Pass 5 as parameter
// end of scope outerFunction
})(7); // Pass 7 as parameter
```

`innerFunction` is closure which is defined inside `outerFunction` and has access to all variable which is declared and defined in outerFunction scope. In addition to this function defined inside function as closure has access to variable which is declared in `global namespace`.

Output of above code would be:

```
outerArg = 7
outerFuncVar = x
innerArg = 5
innerFuncVar = y
globalVar = abc
```

Q14: What will be the output of the following code? ☆☆☆☆

Topics: JavaScript

Problem:

```
var output = (function(x) {
  delete x;
  return x;
})(0);

console.log(output);
```

Solution:

Above code will output `0` as output. `delete` operator is used to delete a property from an object. Here `x` is not an object it's **local variable**. `delete` operator doesn't affect local variable.

Q15: What will be the output of the following code? ☆☆☆☆

Topics: JavaScript

Problem:

```
var Employee = {
  company: 'xyz'
}
var emp1 = Object.create(Employee);
delete emp1.company
console.log(emp1.company);
```

Solution:

Above code will output `xyz` as output. Here `emp1` object got company as **prototype** property. `delete` operator doesn't delete prototype property.

`emp1` object doesn't have **company** as its own property. You can test it like:

```
console.log(emp1.hasOwnProperty('company')); //output : false
```

However, we can delete company property directly from `Employee` object using `delete Employee.company` or we can also delete from `emp1` object using `__proto__` property `delete emp1.__proto__.company`.

Q16: When would you use the `bind` function? ☆☆☆☆

Topics: JavaScript

Answer:

The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

A good use of the `bind` function is when you have a particular function that you want to call with a specific this value. You can then use `bind` to pass a specific object to a function that uses a `this` reference.

```
function fullName() {
  return "Hello, this is " + this.first + " " + this.last;
}

console.log(fullName()); // => Hello this is undefined undefined

// create a person object and pass its values to the fullName function
var person = {first: "Foo", last: "Bar"};
console.log(fullName.bind(person)()); // => Hello this is Foo Bar
```

Q17: How would you add your own method to the Array object so the following code would work? ☆☆☆☆

Topics: JavaScript

Problem:

```
var arr = [1, 2, 3, 4, 5];
var avg = arr.average();
console.log(avg);
```

Solution:

JavaScript is not class based, but it is a prototype-based language. This means that each object is linked to another object, its prototype, and it inherits its methods. You can follow the prototype chain for each object up until you reach the `null` object which has no prototype. We need to add a method to the global `Array` object, and we will do this by modifying the `Array prototype`.

```
Array.prototype.average = function() {
  // calculate sum
  var sum = this.reduce(function(prev, cur) { return prev + cur; });
  // return sum divided by number of elements
  return sum / this.length;
}

var arr = [1, 2, 3, 4, 5];
var avg = arr.average();
console.log(avg); // => 3
```

Q18: What will the following code output? ☆☆☆☆

Topics: JavaScript

Problem:

```
0.1 + 0.2 === 0.3
```

Solution:

This will surprisingly output `false` because of floating point errors in internally representing certain numbers. `0.1 + 0.2` does not nicely come out to `0.3` but instead the result is actually `0.30000000000000004` because the computer cannot internally represent the correct number. One solution to get around this problem is to round the results when doing arithmetic with decimal numbers.

Q19: How would you create a private variable in JavaScript? ☆☆☆☆

Topics: JavaScript

Answer:

To create a private variable in JavaScript that cannot be changed you need to create it as a local variable within a function. Even if the function is executed the variable cannot be accessed outside of the function. For example:

```
function func() {
  var priv = "secret code";
}

console.log(priv); // throws error
```

To access the variable, a helper function would need to be created that returns the private variable.

```
function func() {
  var priv = "secret code";
  return function() {
    return priv;
  }
}

var getPriv = func();
console.log(getPriv()); // => secret code
```

Q20: Explain why the following doesn't work as an IIFE. What needs to be changed to properly make it an IIFE? ☆☆☆☆

Topics: JavaScript

Problem:

```
function foo(){ }();
```

Solution:

IIFE stands for Immediately Invoked Function Expressions. The JavaScript parser reads `function foo(){ }();` as `function foo(){ }` and `();`, where the former is a function declaration and the latter (a pair of brackets) is an attempt at calling a function but there is no name specified, hence it throws `Uncaught SyntaxError: Unexpected token )`.

Here are two ways to fix it that involves adding more brackets: `(function foo(){ })()` and `(function foo(){ }())`. These functions are not exposed in the global scope and you can even omit its name if you do not need to reference itself within the body.

You might also use `void` operator: `void function foo(){ }();`. Unfortunately, there is one issue with such approach. The evaluation of given expression is always `undefined`, so if your IIFE function returns anything, you can't use it. An example:

```
// Don't add JS syntax to this code block to prevent Prettier from formatting it.
const foo = void
function bar() {
    return 'foo';
}();

console.log(foo); // undefined
```