数组

1、找出整型数组中乘积最大的三个数

给定一个包含整数的无序数组，要求找出乘积最大的三个数。

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

computeProduct(unsorted_array); // 21000

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

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

   // Get the product of three largest integers in sorted array
   for (var x = array_n_element; x &gt; array_n_element - 3; x--) {
         product1 = product1 * sorted_array[x];
   }
   product2 = sorted_array[0] * sorted_array[1] * sorted_array[array_n_element];

   if (product1 &gt; product2) return product1;

   return product2
};
```

2、寻找连续数组中的缺失数

给定某无序数组，其包含了 n 个连续数字中的 n - 1 个，已知上下边界，要求以 O(n)的复杂度找出缺失的数字。

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

```
var lower_bound = 1;

findMissingNumber(array_of_integers, upper_bound, lower_bound); //8

function findMissingNumber(array_of_integers, upper_bound, lower_bound) {

   // Iterate through array to find the sum of the numbers
   var sum_of_integers = 0;
   for (var i = 0; i &lt; array_of_integers.length; i++) {
      sum_of_integers += array_of_integers[i];
   }

   //  以高斯求和公式计算理论上的数组和
   // Formula: [(N * (N + 1)) / 2] - [(M * (M - 1)) / 2];
   // N is the upper bound and M is the lower bound

   upper_limit_sum = (upper_bound * (upper_bound + 1)) / 2;
   lower_limit_sum = (lower_bound * (lower_bound - 1)) / 2;

   theoretical_sum = upper_limit_sum - lower_limit_sum;

   //
   return (theoretical_sum - sum_of_integers)
}
```

3、数组去重

给定某无序数组，要求去除数组中的重复数字并且返回新的无重复数组。

```
// 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 &lt; array.length; i++) {
        // If key returns null (unique), it is evaluated as false.
        if(!hashmap.hasOwnProperty([array[i]])) {
            hashmap[array[i]] = 1;
            unique.push(array[i]);
        }
    }
    return unique;
}
```

4、数组中元素最大差值计算

给定某无序数组，求取任意两个元素之间的最大差值，注意，这里要求差值计算中较小的元素下标必须小于较大元素的下标。譬如[7, 8, 4, 9, 9, 15, 3, 1, 10]这个数组的计算值是 11(15 - 4) 而不是 14(15 - 1)，因为 15 的下标小于 1。

```
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) {

    // 如果数组仅有一个元素，则直接返回 -1

    if (array.length &lt;= 1) return -1;

    // current_min 指向当前的最小值

    var current_min = array[0];
    var current_max_difference = 0;

    // 遍历整个数组以求取当前最大差值，如果发现某个最大差值，则将新的值覆盖 current_max_difference
    // 同时也会追踪当前数组中的最小值，从而保证 `largest value in future` - `smallest value before it`

    for (var i = 1; i &lt; array.length; i++) {
```

```
      if (array[i] &gt; current_min &amp;&amp; (array[i] - current_min &gt; current_max_difference)) {
          current_max_difference = array[i] - current_min;
      } else if (array[i] &lt;= current_min) {
          current_min = array[i];
      }
   }

   // If negative or 0, there is no largest difference
   if (current_max_difference &lt;= 0) return -1;

   return current_max_difference;
}
```

5、数组中元素乘积

给定某无序数组，要求返回新数组 output ，其中 output[i] 为原数组中除了下标为 i 的元素之外的元素乘积，要求以 O(n) 复杂度实现：

```
var firstArray = [2, 2, 4, 1];
var secondArray = [0, 0, 0, 2];
var thirdArray = [-2, -2, -3, 2];

productExceptSelf(firstArray); // [8, 8, 4, 16]
productExceptSelf(secondArray); // [0, 0, 0, 0]
productExceptSelf(thirdArray); // [12, 12, 8, -12]

function productExceptSelf(numArray) {
   var product = 1;
   var size = numArray.length;
   var output = [];

   // From first array: [1, 2, 4, 16]
   // The last number in this case is already in the right spot (allows for us)
   // to just multiply by 1 in the next step.
   // This step essentially gets the product to the left of the index at index + 1
   for (var x = 0; x &lt; size; x++) {
       output.push(product);
       product = product * numArray[x];
   }

   // From the back, we multiply the current output element (which represents the product
```

```
    // on the left of the index, and multiplies it by the product on the right of the element)
    var product = 1;
    for (var i = size - 1; i &gt; -1; i--) {
            output[i] = output[i] * product;
            product = product * numArray[i];
    }

    return output;
}
```

6、数组交集

给定两个数组，要求求出两个数组的交集，注意，交集中的元素应该是唯一的。

```
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)
}
```

字符串

1、颠倒字符串

给定某个字符串，要求将其中单词倒转之后然后输出，譬如"Welcome to this Javascript Guide!" 应该输出为 "emocleW ot siht tpircsavaJ !ediuG"。

```
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);
}
```

2、乱序同字母字符串

给定两个字符串，判断是否颠倒字母而成的字符串，譬如 Mary 与 Army 就是同字母而顺序颠倒：

```
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;
}
```

3、回文字符串

判断某个字符串是否为回文字符串，譬如 racecar 与 race car 都是回文字符串：

```
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("");
}
```

栈与队列

1、使用两个栈实现入队与出队

```
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 &lt;= 0) {
        while(stackInput.length &gt; 0) {
            var elementToOutput = stackInput.pop();
            stackOutput.push(elementToOutput);
```

```
        }
    }

    return stackOutput.pop();
}
```

2、判断大括号是否闭合

创建一个函数来判断给定的表达式中的大括号是否闭合：

```
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 &lt;= 0) return true;

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

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

递归

1、二进制转换

通过某个递归函数将输入的数字转化为二进制字符串：

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

function decimalToBinary(digit) {
   if(digit &gt;= 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 '';
   }
}
```

2、二分搜索

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

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

数字

1、判断是否为 2 的指数值

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

// For the non-zero case:
function isPowerOfTwo(number) {
   // `&amp;amp;` uses the bitwise n.
   // In the case of number = 4; the expression would be identical to:
   // `return (4 &amp;amp; 3 === 0)`
   // In bitwise, 4 is 100, and 3 is 011. Using &amp;amp;, 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 &amp;amp; 4 === 0)`, it would be false
   // since it returns 101 &amp;amp; 100 = 100 (NOT === 0)

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

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