```javascript
function isValid(s) { // Function to check if the brackets in the string are valid
  const stack = []; // Stack to keep track of opening brackets
  const map = {     // Mapping of closing brackets to their corresponding opening brackets
    ')': '(',
    '}': '{',
    ']': '['
  };

  for (let char of s) { // Iterate through each character in the string
    if (['(', '{', '['].includes(char)) { // If it's an opening bracket
      stack.push(char); // Push it onto the stack
    } else { // If it's a closing bracket
      if (stack.pop() !== map[char]) return false; // Pop from stack and check if it matches the corresponding opening bracket
    }
  }

  return stack.length === 0; // If stack is empty, all brackets are matched and valid
}
```

```javascript
// The count-and-say sequence is a sequence of digit strings defined by the recursive formula:

// countAndSay(1) = "1"
// countAndSay(n) is the run-length encoding of countAndSay(n - 1).
// Run-length encoding (RLE) is a string compression method that works
// by replacing consecutive identical characters (repeated 2 or more times)
//  with the concatenation of the character and the number marking the count
//  of the characters (length of the run). For example, to compress the string
//  "3322251" we replace "33" with "23", replace "222" with "32", replace "5" with
//  "15" and replace "1" with "11". Thus the compressed string becomes "23321511".

const countAndSay = (n) => {
  if (n === 1) return "1"; // Base case: the first term is always "1"

  let prev = countAndSay(n - 1); // Recursively get the previous term in the sequence
  let result = "";               // Initialize the result string for this term
  let count = 1;                 // Initialize a counter for consecutive digits

  // Loop through the previous term to build the current term
  for (let i = 0; i < prev.length; i++) {
    if (prev[i] === prev[i + 1]) {
      count++; // If the current digit is the same as the next, increment the count
    } else {
      result += count + prev[i]; // Otherwise, append the count and the digit to the result
      count = 1;                 // Reset the count for the next group of digits
    }
  }

  return result; // Return the constructed term
}
```

```javascript
const convertToTtle = (column) => {
  let title = '';
  while (column > 0) {
    column--; // Adjust for 0-based index
    const remainder = column % 26;

    // console.log('fromCahrCode '+ String.fromCharCode(65 + remainder)); // Convert to character
//     The ASCII code for 'A' is 65, 'B' is 66, ..., 'Z' is 90.
// So, if remainder is 0, you get 'A'; if remainder is 1, you get 'B'; ..., if remainder is 25, you get 'Z'.
//     title = String.fromCharCode(65 + remainder) + title; // Convert to character
    // console.log(`Column: ${column}, Remainder: ${remainder}, Title: ${title}`);
    column = Math.floor(column / 26); // Move to the next "digit"
    // console.log(`Next Column: ${column}`);
  }
  return title;
}
// fromCahrCode B
// Column: 27, Remainder: 1, Title: B
// Next Column: 1
// fromCahrCode A
// Column: 0, Remainder: 0, Title: AB
// Next Column: 0
// AB
// 28 - 1 = 27; 27 % 26 = 1 → 'B'; 27 / 26 = 1
// 1 - 1 = 0; 0 % 26 = 0 → 'A'; 0 / 26 = 0
// Result: "AB"
```

```javascript
const strStr = (haystack, needle) => {
  if (needle === "") return 0; // If the needle is an empty string, return 0 (by convention)
  if (haystack.length < needle.length) return -1; // If haystack is shorter than needle, needle can't be found

  // Loop through haystack, stopping so that there's enough room for needle to fit
  for (let i = 0; i <= haystack.length - needle.length; i++) {
    // Check if the substring of haystack starting at i matches needle
    console.log(`Checking substring: ${haystack.substring(i, i + needle.length)} against needle: ${needle}`);
    if (haystack.substring(i, i + needle.length) === needle) {
      return i; // If found, return the starting index
    }
  }

  return -1; // If needle is not found in haystack, return -1
}

// Example usage:
console.log(strStr("hello", "ll")); // Output: 2
```

```javascript
  // Problem: Given two strings, jewels and stones, where
  // each character in jewels represents a type of jewel
  // and each character in stones represents a stone you have.
  // Return how many stones you have are also jewels.
  const jewelAndStones = (jewels, stones) => {
    // Create a set to store the jewels for O(1) lookup
    const jewelSet = new Set(jewels);
    console.log(jewelSet); // Debugging line to see the contents of the jewel set
    // Initialize a counter for the number of stones that are jewels
    let count = 0;

    // Iterate through each stone you have
    for (const stone of stones) {
      // If the stone is a jewel, increment the count
      if (jewelSet.has(stone)) {
        count++;
      }
    }
    // Return the total count of stones that are jewels
    return count;
  }
```

```
// Given a string s, find the first non-repeating character
// in it and return its index. If it does not exist, return -1.
// Example 1:// Input: s = "leetcode"// Output: 0
// Explanation:// The character 'l' at index 0 is the first character that does not occur at any other index.
const firstUniqueCharacter = (s) => {
 const charCount = {};
  // Count the occurrences of each character
  for (let char of s) {
    charCount[char] = (charCount[char] || 0) + 1;
  }
  console.log(charCount);
  // Find the first unique character
  for (let i = 0; i < s.length; i++) {
    console.log(s[i], charCount[s[i]]);
    if (charCount[s[i]] === 1) {
      return i;
    }
  }
  return -1; // If no unique character found
}
console.log(firstUniqueCharacter("leetcode")); // Output: 0
```

```
  // Write a function to find the longest common prefix string amongst an array of strings.
  // If there is no common prefix, return an empty string "".
  // Example 1:// Input: strs = ["flower","flow","flight"]// Output: "fl"
  // Example 2:// Input: strs = ["dog","racecar","car"]
  // Output: ""// Explanation: There is no common prefix among the input strings.
  const longestCommonPrefix = (strs) => { // Define a function that takes an array of strings
      if (strs.length === 0) return ""; // If the array is empty, return an empty string

      let prefix = strs[0]; // Start with the first string as the initial prefix

      for (let i = 1; i < strs.length; i++) { // Loop through each string in the array starting from the second
          while (strs[i].indexOf(prefix) !== 0) { // While the current prefix is not a prefix of the current string
              prefix = prefix.slice(0, -1); // Shorten the prefix by removing the last character
              if (prefix === "") return ""; // If the prefix becomes empty, return an empty string
          }
      }
      return prefix; // Return the longest common prefix found
  }
```

```
function palindromeStr(s) {
    if (s == reverse(s)) { return true;}
    return false;
}
function reverse(s) {
    newStr = '';
    //console.log(s.length)
    for (let i = s.length - 1; i >= 0; i--) {
        newStr += s[i]
    }
    return newStr;
}
const longestSubstringWithoutRepeatingCharacters = (s) => { // Define a function that takes a string s
    let seen = {}                    // Object to store the last seen index of each character
    let start = 0;                   // Start index of the current substring window
    let maxLength = 0;               // Maximum length of substring found so far

    for(let end = 0; end < s.length; end++) { // Loop through each character in the string with 'end' as the end index
        let char = s[end];           // Current character at position 'end'

        // If the character is already seen and is within the current window
        if (seen[char] >= start) {   // If the character was seen and its last index is within the current window
            start = seen[char] + 1;  // Move the start to one position after the last occurrence of the character
        }

        // Update the last seen index of the character
        seen[char] = end;            // Record the current index for the character

        // Calculate the length of the current substring
        let currentLength = end - start + 1; // Length of the current window without repeating characters
        // Update the maximum length if the current length is greater
        maxLength = Math.max(maxLength, currentLength); // Update maxLength if currentLength is larger
    }
    return maxLength;                // Return the length of the longest substring without repeating characters
}
```

```javascript
const reversePolishNotation = (tokens) => {
  const stack = [];

  for (const token of tokens) {
    if (!isNaN(token)) {
      stack.push(Number(token));
    } else {
      const b = stack.pop();
      const a = stack.pop();

      switch (token) {
        case '+':
          stack.push(a + b);
          break;
        case '-':
          stack.push(a - b);
          break;
        case '*':
          stack.push(a * b);
          break;
        case '/':
          stack.push(Math.trunc(a / b)); // Use Math.trunc to handle integer division
          break;
      }
    }
  }

  return stack[0];
}
```

```javascript
const reverseString = (str) => {
    if (typeof str !== 'string') {
        throw new Error('Input must be a string');
    }
    let reversed = '';
    for (let i=str.length - 1; i >= 0; i--) {
        reversed += str[i];
    }
    return reversed;
}
```

```javascript
const reverseWords = (str) => {
    let reversed = '';
    let splitString = str.split(" "); // Split the string by spaces

    for (let i = splitString.length - 1; i >= 0; i--) {
        if (splitString[i] !== "") { // Skip empty strings caused by multiple spaces
            reversed += splitString[i] + " ";
        }
    }

    return reversed.trim(); // Remove the trailing space
}

// Example usage:
console.log(reverseWords("hello world")); // "world hello"
console.log(reverseWords("  a good   example  ")); // "example good a"

// Example usage:
console.log(reverseWords("hello world")); // "world hello"
```