Functions, Substrings, and Subarrays in JavaScript

JavaScript Functions

Functions in JavaScript are reusable blocks of code designed to perform specific tasks. They help in improving code readability, modularity, and reducing redundancy.

Defining a Function

A function is defined using the `function` keyword, followed by a name, parentheses `()` for parameters, and a block of code within curly braces `{}`.

```
function greet(name) {
    // This function greets the person passed as a parameter.
    console.log("Hello, " + name + "!");
}
```

Calling a Function

To execute a function, use its name followed by parentheses `()` and pass the required arguments.

```
greet("Alice"); // Output: Hello, Alice!
```

Return Statement

The `return` statement allows functions to return a value that can be used elsewhere.

```
function square(x) {
    // This function returns the square of the input.
    return x * x;
}

let result = square(5);
console.log(result); // Output: 25
```

Function Parameters

Functions can take one or more parameters, separated by commas.

```
function add(a, b) {
    // Add two numbers and return the result.
    return a + b;
}

console.log(add(3, 4)); // Output: 7
```

Default Parameters

Default parameter values are used if no argument is provided for the corresponding parameter.

```
function power(base, exponent = 2) {
    // Raise base to the power of exponent.
    return Math.pow(base, exponent);
}

console.log(power(2));    // Output: 4 (2^2)
console.log(power(2, 3)); // Output: 8 (2^3)
```

Example: Check for Prime Number

A function to check if a given number is prime:

```
function isPrime(num) {
    // Check if a number is prime.
    if (num <= 1) return false;
    for (let i = 2; i <= Math.sqrt(num); i++) {
        if (num % i === 0) return false;
    }
    return true;
}

console.log(isPrime(17)); // Output: true
console.log(isPrime(21)); // Output: false
```

Advanced Topics in Functions

Arrow Functions

Arrow functions provide a shorter syntax for writing functions:

```
const greet = (name) => console.log("Hello, " + name + "!");
greet("Bob"); // Output: Hello, Bob!
```

Array Methods

Arrow functions simplify operations with methods like `map`, `filter`, and `reduce`.

- Shorter syntax
- Implicit return when using a single expression
- Doesn't bind its own `this` value

- Great for one-liner functions

```
// Traditional function
function add(a, b) {
    return a + b;
}

// Same function as an arrow function
const add = (a, b) => a + b;

const numbers = [1, 2, 3, 4];
```

Arrow Functions with Array.map():

```
// Traditional way
const doubled = numbers.map(function(num) {
    return num * 2;
});

// With arrow function - much cleaner!
const doubled = numbers.map(num => num * 2);
// Result: [2, 4, 6, 8]

const numbers = [1, 2, 3, 4, 5, 6];
```

Arrow Functions with Array.filter():

```
// Traditional way
const evenNumbers = numbers.filter(function(num) {
    return num % 2 === 0;
```

```
});

// With arrow function
const evenNumbers = numbers.filter(num => num % 2 === 0);
// Result: [2, 4, 6]
```

Arrow Functions with Array.reduce():

```
const numbers = [1, 2, 3, 4];

// Traditional way
const sum = numbers.reduce(function(total, current) {
    return total + current;
}, 0);

// With arrow function
const sum = numbers.reduce((total, current) => total + current, 0);
// Result: 10
const products = [
    { name: 'Laptop', price: 1000 },
    { name: 'Phone', price: 500 },
    { name: 'Tablet', price: 300 },
    { name: 'Watch', price: 200 }
];

// Let's find the total price of products over $300
```

```
const expensiveProductsTotal = products
    .filter(product => product.price > 300) // First filter expensive products
    .map(product => product.price)          // Extract just the prices
    .reduce((total, price) => total + price, 0); // Sum them up


console.log(expensiveProductsTotal); // Result: 1500
```

Anonymous Functions

Functions without a name are called anonymous functions and are often used as arguments to other functions:

```
let add = function (n1,n2){

    return n1+n2

}



let result= add(5,6)

console.log(result)
```

Callback Functions

A function passed as an argument to another function is called a callback:

```
function processInput(input, callback) {
    console.log("Processing input: " + input);
    callback();
}

processInput("Data", function() {
    console.log("Callback executed");
});

// Main function that takes array and callback function as parameters
function processNumbers(numbers, callback) {
    // Process each number and collect results
    const results = [];

    for(let num of numbers) {
```

```
        // Apply the callback to each number
        const result = callback(num);
        results.push(result);
    }

    return results;
}

// Different callback functions we can use
function double(num) {
    return num * 2;
}

function addTen(num) {
    return num + 10;
}

// Using callbacks
const myNumbers = [1, 2, 3, 4];

const doubledNumbers = processNumbers(myNumbers, double);
console.log('Doubled:', doubledNumbers); // [2, 4, 6, 8]

const numbersPlusTen = processNumbers(myNumbers, addTen);
console.log('Plus ten:', numbersPlusTen); // [11, 12, 13, 14]

// Using an anonymous callback function
const squaredNumbers = processNumbers(myNumbers, function(num) {
    return num * num;
});
console.log('Squared:', squaredNumbers); // [1, 4, 9, 16]
```

Additional Problems

1. **Find the Maximum of Three Numbers**:
```
function maxOfThree(a, b, c) {
    return Math.max(a, b, c);
}
console.log(maxOfThree(10, 20, 15)); // Output: 20
```
2. **Factorial Using Recursion**:
```
function factorial(n) {
    if (n === 0) return 1;
    return n * factorial(n - 1);
}
console.log(factorial(5)); // Output: 120
```
3. **Reverse a String**:
```
function reverseString(str) {
    return str.split('').reverse().join('');
}
console.log(reverseString("hello")); // Output: olleh
```

Subarrays in JavaScript

A subarray is a contiguous sequence of elements from an array. JavaScript uses the `slice` method to extract subarrays.

Creating Subarrays

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
let sub1 = arr.slice(2, 5); // Elements from index 2 to 4
console.log(sub1); // Output: [3, 4, 5]
```

Example: Generate All Subarrays

```
function generateSubarrays(arr) {
    let n = arr.length;
    console.log("All Non-empty Subarrays:");
    for (let i = 0; i < n; i++) {
        for (let j = i; j < n; j++) {
            let subarray = arr.slice(i, j + 1);
            console.log(subarray);
        }
    }
}

let arr = [1, 2, 3, 4];
generateSubarrays(arr);
```

Substrings in JavaScript

A substring is a contiguous sequence of characters within a string. You can extract substrings using the `slice`, `substring`, or `substr` methods.

Creating Substrings

```
let s = "Hello, World!";
let substring1 = s.slice(7, 12); // Characters from index 7 to 11
console.log(substring1); // Output: World

let substring2 = s.slice(0, 5); // Characters from beginning to index 4
console.log(substring2); // Output: Hello
```

Example: Generate All Substrings

```
function generateSubstrings(str) {
    let n = str.length;
    console.log("All Substrings:");
    for (let i = 0; i < n; i++) {
        for (let j = i + 1; j <= n; j++) {
            console.log(str.slice(i, j));
        }
    }
}

generateSubstrings("abc");
```

Missing Points About Functions

- **Rest Parameters**: Allows functions to accept an indefinite number of arguments.
- `function sum(...numbers) {`
- `    return numbers.reduce((a, b) => a + b, 0);`
- `}`
- `console.log(sum(1, 2, 3, 4)); // Output: 10`

- **Function Expressions**: Functions can be assigned to variables.
- `const multiply = function(a, b) {`
- `    return a * b;`
- `};`
- `console.log(multiply(3, 4)); // Output: 12`

Leetcode problem on subarray

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

1. Brute Force Solution

```
function maxSubArrayBruteForce(nums) {
  let maxSum = -Infinity; // Initialize to the smallest possible number

  for (let i = 0; i < nums.length; i++) {
    let currentSum = 0; // Sum of the subarray starting at index `i`

    for (let j = i; j < nums.length; j++) {
      currentSum += nums[j]; // Add the current element to the subarray sum
      maxSum = Math.max(maxSum, currentSum); // Update maxSum if
currentSum is larger
    }
  }

  return maxSum;
}

// Example usage:
const nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
console.log(maxSubArrayBruteForce(nums)); // Output: 6
```

2. Optimal solution (Kadanes algorithm)

```
function maxSubArrayKadane(nums) {
  let maxSum = nums[0]; // Initialize maxSum to the first element
  let currentSum = nums[0]; // Initialize currentSum to the first element

  for (let i = 1; i < nums.length; i++) {
    // Either add the current number to the existing subarray or start a new subarray
    currentSum = Math.max(nums[i], currentSum + nums[i]);

    // Update maxSum if currentSum is greater
    maxSum = Math.max(maxSum, currentSum);
  }
```

finding out indexes of maxsum subarray

```
function maxSubArray(nums) {
  let maxSum = nums[0];  // Initialize max sum to the first element
  let currentSum = nums[0];  // Initialize current sum to the first element
  let start = 0;  // The starting index of the subarray with max sum
  let end = 0;  // The ending index of the subarray with max sum
  let tempStart = 0;  // Temporary variable to track the potential starting index

  for (let i = 1; i < nums.length; i++) {
    const num = nums[i];

    // If current number itself is greater than adding it to currentSum, reset
    if (num > currentSum + num) {
      currentSum = num;
      tempStart = i;  // Start a new subarray at the current index
    } else {
      currentSum += num;
    }
```

```
      // Update maxSum and the indices if we find a larger sum
      if (currentSum > maxSum) {
        maxSum = currentSum;
        start = tempStart;  // Update start index of the subarray
        end = i;  // Update end index of the subarray
      }
    }

    // Output the maximum sum and the indices of the subarray
    console.log("Maximum Sum:", maxSum);
    console.log("Subarray indices:", start, "to", end);
    console.log("Subarray:", nums.slice(start, end + 1)); // To print the subarray
    return maxSum;
}

// Example usage:
const nums = [1, -2, 3, 4, -1, 2, 1];
maxSubArray(nums);
```