Strings Javascript-1

Reading Material

PW SKILLS

Pre-requisites:

Basics of Javascript

Topics to learn

- What are Strings
- How Strings work in javascript
- Reversing a string
- Finding a substring
- Using RegEx to find patterns
- Find and replace portion of string
- Leetcode 125: Valid Palindrome
- Leetcode 945: Minimum increment to make array unique
- Leetcode 131: Palindrome partitioning

What are Strings?

Strings are a sequence of characters used to represent text. In JavaScript, strings are sequences of Unicode characters enclosed in single quotes (`'`), double quotes (`"`), or backticks (`` ` ``). Strings are immutable, meaning once a string is created, its value cannot be changed. Instead, any modification to a string results in the creation of a new string.

How Strings Work in JavaScript

Declaration and Initialization:
You can declare a string using single quotes, double quotes, or backticks. Backticks are used for template literals, allowing for multi-line strings and embedded expressions.

```
let str1 = 'Hello, World!';
let str2 = "Hello, World!";
let str3 = `Hello, World!`;
```

Properties and Methods:

JavaScript strings have various properties and methods to manipulate and access text.

- **Length Property:** Returns the length of the string.

```
let str = "Hello, World!";
console.log(str.length);  // Output: 13
```

Accessing Characters: Use bracket notation or the charAt() method.

```
console.log(str[0]);       // Output: H
console.log(str.charAt(0)); // Output: H
```

Concatenation: Combine strings using the + operator or template literals.

```
let str1 = "Hello";
let str2 = "World";
let greeting = str1 + " " + str2;
console.log(greeting);  // Output: Hello World

// Using template literals
let message = `${str1}, ${str2}!`;
console.log(message);  // Output: Hello, World!
```

Escape Sequences:

Use backslashes to escape special characters like newlines, quotes, or tabs.

```
let quote = "He said, \"JavaScript is fun!\"";
console.log(quote);  // Output: He said, "JavaScript is
fun!"

let multiline = "Line 1\nLine 2\nLine 3";
console.log(multiline);
// Output:
// Line 1
// Line 2
// Line 3
```

Basic String Operations

Concatenation:
Combine two or more strings.

```
let firstName = "John";
let lastName = "Doe";
let fullName = firstName + " " + lastName;
console.log(fullName);  // Output: John Doe
```

Template Literals:

Use backticks for embedded expressions and multi-line strings.

```
let age = 25;
let message = `Hello, my name is ${firstName} ${lastName}
and I am ${age} years old.`;
console.log(message);  // Output: Hello, my name is John
Doe and I am 25 years old.
```

Searching and Extracting Substrings:

Use indexOf(), includes(), slice(), substring(), and split().

```
let str = "Hello, World!";
console.log(str.indexOf("World"));      // Output: 7
console.log(str.includes("World"));     // Output: true
console.log(str.slice(0, 5));           // Output: Hello
console.log(str.substring(7, 12));      // Output: World
console.log(str.split(" "));            // Output: [
'Hello,', 'World!' ]
```

Replacing Parts of a String:

Use replace() for simple replacements and replaceAll() for global replacements.

```
let str = "The quick brown fox jumps over the lazy dog.";
let newStr = str.replace("fox", "cat");
console.log(newStr);  // Output: The quick brown cat jumps
over the lazy dog.

let text = "JavaScript is fun. JavaScript is versatile.";
let updatedText = text.replaceAll("JavaScript", "JS");
console.log(updatedText);  // Output: JS is fun. JS is
versatile.
```

String Methods:

JavaScript provides various string methods for manipulation:

- toUpperCase() / toLowerCase():

```
let str = "Hello, World!";
console.log(str.toUpperCase());  // Output: HELLO, WORLD!
console.log(str.toLowerCase());  // Output: hello, world!
```

trim(): Removes whitespace from both ends of a string.

```
let str = "   Hello, World!   ";
console.log(str.trim());  // Output: Hello, World!
```

concat(): Concatenates two or more strings.

```
let str1 = "Hello, ";
let str2 = "World!";
console.log(str1.concat(str2));  // Output: Hello, World!
```

String Template Literals:

Use ${} to embed expressions inside strings.

```
let name = "Alice";
let age = 30;
console.log(`My name is ${name} and I am ${age} years
old.`);  // Output: My name is Alice and I am 30 years old.
```

Reversing a String

Concept: Reversing a string means flipping the sequence of characters so that the last character becomes the first, the second last becomes the second, and so on.

Code Example:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}

const input = "hello";
const reversed = reverseString(input);
console.log(reversed);  // Output: "olleh"
```

Explanation:

- **split('')** converts the string into an array of characters.
- **reverse()** reverses the array.
- **join('')** joins the array back into a string.

Finding a Substring

Concept: A substring is a sequence of characters that appears in the same order within another string.

Code Example:

```
const str = "hello world";
const substring = "world";

const position = str.indexOf(substring);
if (position !== -1) {
    console.log(`Substring found at index ${position}`);
} else {
    console.log("Substring not found");
}
```

Explanation:

- indexOf() returns the index of the first occurrence of the substring, or -1 if the substring is not found.

Using RegEx to Find Patterns (Validation)

Concept: Regular expressions (RegEx) are patterns used to match character combinations in strings. They are essential for text validation and parsing.

Code Example:

```
const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

function validateEmail(email) {
    return emailPattern.test(email);
}

const testEmail = "example@example.com";
console.log(validateEmail(testEmail));  // Output: true
```

Explanation:

- /^[**^\s@]+@[^\s@]+.[^\s@]+$/** is a regular expression to validate email format.
  - ^ asserts the start of the string.
  - [^\s@]+ matches one or more characters that are not whitespace or '@'.
  - @ matches the literal '@' character.
  - [^\s@]+ matches one or more characters that are not whitespace or '@'.
  - \. matches a literal dot '.'.
  - [^\s@]+ matches one or more characters that are not whitespace or '@'.
  - $ asserts the end of the string.

Find and Replace a Portion of a String

Concept: Replacing a part of a string involves substituting a specified substring with another string.

Code Example:

```
const str = "The quick brown fox jumps over the lazy dog";
const newStr = str.replace("fox", "cat");
console.log(newStr);  // Output: "The quick brown cat jumps
over the lazy dog"
```

Explanation:

- replace(oldSubstring, newSubstring) returns a new string with the first occurrence of oldSubstring replaced by newSubstring.
- Use a regular expression with the global flag (/g) to replace all occurrences.

```
const str = "The quick brown fox jumps over the lazy dog";
const newStr = str.replace(/fox/g, "cat");
console.log(newStr);  // Output: "The quick brown cat jumps
over the lazy dog"
```

Leetcode 125. Valid Palindrome

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

Example 1:
Input: s = "A man, a plan, a canal: Panama"
Output: true

Explanation: "amanaplanacanalpanama" is a palindrome.

Example 2:
Input: s = "race a car"
Output: false

Explanation: "raceacar" is not a palindrome.

Example 3:
Input: s = " "
Output: true

Explanation: s is an empty string "" after removing non-alphanumeric characters.
Since an empty string reads the same forward and backward, it is a palindrome.

Constraints:
- 1 <= s.length <= 2 * 105
- s consists only of printable ASCII characters.

Solution

Code:

```
/**
 * @param {string} s
 * @return {boolean}
 */
var isPalindrome = function(s) {
  s=  s.toLowerCase().replace(/[^a-z0-9]/gi,'');
  let rev="";
  for(let i=0;i<s.length;i++){
rev= s[i]+rev
  }
  if(rev==s)return true;

  return false;

};
```

Complexity

- Time complexity: O(n)
- Space complexity: O(n)

Leetcode 945: Minimum Increment to Make Array Unique

You are given an integer array nums. In one move, you can pick an index i where 0 <= i < nums.length and increment nums[i] by 1.

Return the minimum number of moves to make every value in nums unique.
The test cases are generated so that the answer fits in a 32-bit integer.

Example 1:
Input: nums = [1,2,2]
Output: 1

Explanation: After 1 move, the array could be [1, 2, 3].

Example 2:
Input: nums = [3,2,1,2,1,7]
Output: 6

Explanation: After 6 moves, the array could be [3, 4, 1, 2, 5, 7].
It can be shown with 5 or less moves that it is impossible for the array to have all unique values.

Constraints:
- 1 <= nums.length <= 105
- 0 <= nums[i] <= 105

Solution:

Approach

- **Sort the Array:** Begin by sorting the array. This helps in easily identifying consecutive duplicates.
- **Iterate Through the Array:** Start iterating from the second element. For each element, check if it is less than or equal to the previous element.
- **Increment to Make Unique:** If the current element is not greater than the previous element, increment it to make it one more than the previous element. Track the total number of increments made.
- **Return the Result:** After processing all elements, the total count of increments required to make all elements unique is the result.

Code:

```
var minIncrementForUnique = function(nums) {
    if (nums.length === 0) {
        return 0;
    }
```

```
nums.sort((a, b) => a - b);

    let count = 0;


    for (let i = 1; i < nums.length; ++i) {
        if (nums[i] <= nums[i - 1]) {
            let increment = nums[i - 1] - nums[i] + 1;
            nums[i] += increment;
            count += increment;
        }
    }

    return count;
};
```

Complexity

- **Time complexity: O(n log n)**
  -> Sorting the array takes O(nlogn) time.
  -> Iterating through the array takes O(n) time.

- **Space complexity: O(1)**
  -> No extra space is used other than the input array, which is modified in place.

Leetcode 131: Palindrome Partitioning

Given a string s, partition s such that every
substring
of the partition is a
palindrome
. Return all possible palindrome partitioning of s.

Example 1:
Input: s = "aab"
Output: [["a","a","b"],["aa","b"]]

Example 2:
Input: s = "a"
Output: [["a"]]

Constraints:
- 1 <= s.length <= 16
- s contains only lowercase English letters.

Solution:

Approach

1. **Backtracking:** Use a recursive function to generate partitions. At each step, choose a substring, check if it's a palindrome, and then recursively partition the rest of the string.
2. **Palindrome Check:** Implement a helper function to check if a given substring is a palindrome.
3. **Base Case:** When the start index reaches the end of the string, add the current partition to the result.

Code:

```
/**
 * @param {string} s
 * @return {string[][]}
 */
var partition = function(s) {
    const ans = [];
    const candidate = [];

    const is_palindrome = (s, l, r) => {
        while (l < r) {
            if (s[l] !== s[r]) {
                return false;
            }
            l++;
            r--;
        }
        return true;
    }

    const solve = (s, i, previous) => {
        if (i === s.length) {
            if (is_palindrome(s, previous, i)) {
                ans.push([...candidate]);
            }
        } else {
            if (is_palindrome(s, previous, i)) {
                candidate.push(s.substring(previous, i+1));
 solve(s, i + 1, i + 1);
                candidate.pop();
            }

            solve(s, i + 1, previous);
        }
    }

    solve(s, 0, 0);
    return ans;
};
```

Complexity

- **Time complexity:** The time complexity is O(N2), where N is the length of the string. This is due to the recursive exploration of all possible partitions and the palindrome checks for each substring.
- **Space complexity:**The space complexity is O(N) for the recursion stack and current partition storage