H1 Recursion with Strings

H2 Important Rule for Recursion with Strings

When it comes to any recursion when dealing with a strings like - where you *most likely deal with the first character* of the string

- Remove Duplicate Character (first character of the string)
- Remove spaces (first character of the string)

Then the recursive case can be simplified with the following formula

```
recursiveFn(string) {
  ...
  ...
  ...
  // recursive case
  return string[0] + recursiveFn(string.substr(1));
}
```

H2 i. Remove Spaces and Tabs in a String

Write a function to remove spaces and tabs from a given string - recursively

H3 Approach 01

- One Approach is to convert the String into a Character Array and then apply the recursive function

```
// remove space
const removeSpace = (arr, result = []) => {
    if (arr.length == 0) return result;
    const char = arr.shift();
    if (char !== ' ' || char !== '\t') result.push(char);
    return removeSpace(arr, result);
};

const removeSpaceExecution = (inputString) => {
    const result = removeSpace(inputString.split(''));
    return result.join('');
};

console.log(`Remove space`, removeSpaceExecution('Hi There'));
```

H3 Approach 02

Second Approach is to apply the recursive function directly on the string using concatination

Removing tabs from a null string `""` will just return the null string `""`. Therefore, this will be our **base case**.

For the **recursive case**, we check whether or not the *first* element is `"\t"` or `" "`. Either the string passed as the argument starts with a tab or space, or it doesn't. If it starts with a tab, then the answer is equal to the string resulting from the removal of all tabs from the second character onwards. Otherwise, the result is the concatenation of the first character in the string, and the string resulting from the removal of all tabs from the rest of the string.

If the first element is `"\t"` or `" "` we discard it and call another instance of the function after removing that element.

1. If the first element is not `"\t"` or `" "`, we append it to the output string and call another instance of the function after removing that element.

```
function remove(string) {
  // Base case
  if (string.length == 0) {
      return "";
  }

  // Recursive case
  if (string[0] == "\t" || string[0] == " ") {
      return remove(string.substr(1));
  }
  else {
      return string[0] + remove(string.substr(1));
  }
}

// Driver Code
var myString = "Hello\tWorld";
console.log(remove(myString));
```

H2 ii. Remove all Adjacent Duplicates from a String

This means that we'll remove all extra instances of a character when multiple instances are found together. In other words, only one instance should remain after this process.

Solution

To remove duplicates, we reduce the length of the string with each recursive call. If the current character is similar to the following character, we discard its first occurrence and repeat the process recursively on the rest of the string. However, if the current character is not similar to the following character, we keep it. We append it when the child function returns.

The code snippet above checks for 33 cases:

1. **(Line number 3 to 5)** If the length of the string is less than or equal to 11.There can be no duplicates in a string that contains only 11 character or no characters so we return the string variable in this case.

> *Condition 11 is the* ***base case*** *since there is no string manipulation occurring in this case.*

1. **(Line number 8 to 10)** If the first and second elements of the string are same.
    - In this case, we discard the first occurrence and recursively call the same function with the first letter removed.
2. **(Line number 13)** If none of these 22 conditions are satisfied, we save the first element for later use and call the same function again with the first element removed. The saved element will be later appended to the beginning of the string when the recursive call returns.

```
function removeDuplicates(string) {
  // Base case
  if (string.length <= 1) {
      return string;
  }

  // Recursive case1
  else if (string[0] == string[1]) {
      return removeDuplicates(string.substr(1));
  }

  // Recursive case2
  return string[0] + removeDuplicates(string.substr(1));
}

// Driver Code
console.log(removeDuplicates('Hellloo'));
```

iii. Merge two sorted Strings Lexicographically

Lexicographical means that something is organized according to alphabetical order.

> *Lower case letters are different from upper case letters and are therefore treated as different elements. All upper case letters come before lower case letters. Alphabetic sorting is as follows: $A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z$*

H3 Conditions

- All Upper case letters come before the lower case letters
- Both the input strings are already sorted
- Alphabetic sorting is as follows: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

H3 Approach

Merging 2 sorted strings Lexicographically has a special formula

- It has 3 base case scenarios
  - 1 - If Both the strings are equal, `return ""` ;
  - 2 & 3 - If Either strings are equal to empty string, return the other
- It has 2 recursive case scenarios
  - Compare the first character of the string with other
    - Call the recursive method for a string formula

```
recursiveFn(string) {
  ...
  ...
  ...
  // recursive case
  return string[0] + recursiveFn(string.substr(1));
}
```

The above approach can be simplified by the following notation

$$mergeLex(s1, s2) = \begin{vmatrix} s1 == s2 & return\ "" \\ s1 == "" & return\ s2 \\ s2 == "" & return\ s1 \\ s1[0] > s2[0] & return\ s2[0] + mergeLex(s1, s2.substr(1)) \\ s2[0] > s1[0] & return\ s1[0] + mergeLex(s1.substr(1), s2) \end{vmatrix}$$

H3 Solution

The above notation can be coded as follows

```
// Merge two strings Lexicographically
const mergeLex = (s1, s2) => {
```

```
  // base case 01
  if (s1 === s2) return "";
  // base case 02
  else if (s1 === '') return s2;
  // base case 03
  else if (s2 === '') return s1;
  // recursive case 01
  else if (s1[0] > s2[0]) return s2[0] + mergeLex(s1, s2.substr(1));
  // recursive case 02
  else { return s1[0] + mergeLex(s1.substr(1), s2); }
}

const string1 = `abcd`;
const string2 = `DEFGHI`;

console.log('Merging of 2 sorted strings - lexi is: ', lex(string1,
string2));
```

H2 iv. Using Recursion, Find the length of the string

Implement a function that takes a string `testVariable` and returns the length of the string.

> *Try to solve this problem using recursion.*

H3 Input

A variable `testVariable` that contains a string.

- `abcdefg`
- `pramod`
- `shwetha`

H3 Output

The length of the input string - 7, 6, 7

H3 Approach 01

We use the same formula of String recursion, except we modify it slightly since we need to keep a counter running

- Base case: If the length of the string equals 0, return the counter
- Else increment the counter (instead of adding it to the string recursion formula)
- Call the recursive function (with a diminishing condition)

H3 Solution 01

The above approach can be coded as follows

```
// Merge two strings Lexicographically
const strLen = (s1, count=0) => {
  if (s1.length === 0) return count;
  else {
    count++;
    return strLen(s1.substr(1), count);
  }
}

const string1 = `abcd`;
const string2 = `DEFGHI`;

console.log('Length of the string is: ', strLen(string2));
```

H3 Approach 02

We use exactly the same formula for ***String recursion***

```
recursiveFn(string) {
  ...
  // recursive case
  return string[0] + recursiveFn(string.substr(1));
}
```

H3 Solution 02

The above approach can be coded as follows

```
function recursiveLength(testVariable) {
  // Base case - same as approach 01
  if (testVariable === "") {
    return 0;
  }

  // Recursive case - using string recursion formula
  else {
    return 1 + recursiveLength(testVariable.substr(1));
  }
}

// Driver Code
testVariable = "Educative";
console.log(recursiveLength(testVariable));
```

H2 v. Sum of Digits in a String

Given a string containing numbers, calculate the sum of those numbers

H3 Approach

We use exactly the same formula for ***String recursion***

```
recursiveFn(string) {
  ...
  // recursive case
  return string[0] + recursiveFn(string.substr(1));
}
```

H3 Solution

The above approach can be coded as follows

```
function sumOfDigits(testVariable) {
  // Base case
  if (testVariable === "") {
    return 0;
  }

  // Recursive case - using string recursion formula
  else {
    return Number(testVariable[0]) +
sumOfDigits(testVariable.substr(1));
  }
}

// Driver Code
testVariable = "1234567";
console.log(`Sum of Digits of ${testVariable}`,
sumOfDigits(testVariable));
```

H2 vi. Check if Palindrome