1) Print Duplicate characters from a string.

What to do (WTD): Traverse through the given string and identify characters that appear more than once.
(e.g.: I/P: "apple" ,O/P: "p")

```c
#include <stdio.h>
#include <string.h>

void findDuplicateCharacters(const char *input_string) {
    int char_count[256] = {0};  // Initialize an array to store character counts

    // Iterate through the string
    for (int i = 0; i < strlen(input_string); i++) {
        char_count[(int)input_string[i]]++;  // Increment the count for the current character
    }

    // Iterate through the character counts
    printf("Duplicate characters: ");
    for (int i = 0; i < 256; i++) {
        if (char_count[i] > 1) {
            printf("%c ", (char)i);  // Print characters with count > 1
        }
    }
    printf("\n");
}

int main() {
    const char *input_str = "apple";
    findDuplicateCharacters(input_str);
    return 0;
}
```

2) Convert a string to its integer representation without using any built-in functions.

WTD: Transform a given string of numeric characters into its corresponding integer representation without relying on built-in methods.
(e.g., "1234" to 1234)

```c
#include <stdio.h>

int stringToInteger(const char *str) {
    int result = 0;
    int sign = 1;  // To handle negative numbers

    // Check for a negative sign
    if (*str == '-') {
        sign = -1;
        str++;  // Move past the negative sign
    }

    // Iterate through the characters in the string
    while (*str != '\0') {
        if (*str >= '0' && *str <= '9') {
            // Convert the character to its integer value and update the
result
            result = result * 10 + (*str - '0');
        } else {
            // If a non-numeric character is encountered, stop the
conversion
            break;
        }
        str++;  // Move to the next character
    }

    // Apply the sign to the result
    return result * sign;
}

int main() {
```

```
    const char *input_str = "1234";
    int result = stringToInteger(input_str);
    printf("Integer representation: %d\n", result);
    return 0;
}
```

3) Print the first non-repeated character from a string.

WTD: Examine the string and pinpoint the very first character that doesn't repeat elsewhere in the string.
(e.g.: I/P: "swiss" ,O/P: "w")

```
#include <stdio.h>
#include <string.h>

char firstNonRepeatedCharacter(const char *str) {
    int char_count[256] = {0};  // Initialize an array to store character counts

    // Iterate through the string and count the occurrences of each character
    for (int i = 0; i < strlen(str); i++) {
        char_count[(int)str[i]]++;
    }

    // Iterate through the string again to find the first non-repeated character
    for (int i = 0; i < strlen(str); i++) {
        if (char_count[(int)str[i]] == 1) {
            return str[i];  // Return the first non-repeated character
        }
    }

    // If no non-repeated character is found, return a sentinel value (e.g., '\0')
    return '\0';
```

```
}

int main() {
    const char *input_str = "swiss";
    char result = firstNonRepeatedCharacter(input_str);

    if (result != '\0') {
        printf("First non-repeated character: %c\n", result);
    } else {
        printf("No non-repeated character found.\n");
    }

    return 0;
}
```

4) Find the longest palindrome substring in a given string.

WTD: Identify the longest continuous sequence within the string that reads the same backward as forward.
(e.g.: I/P: "babad" ,O/P: "bab")

```
#include <stdio.h>
#include <string.h>

// Function to find the longest palindrome substring in a given string
char* longestPalindrome(char* s) {
    int len = strlen(s);
    if (len <= 1) {
        return s; // A single character or an empty string is a palindrome
    }

    // Create a table to store the palindrome information
    int table[len][len];
    memset(table, 0, sizeof(table));

    int max_len = 1; // Initialize the maximum palindrome length
    int start = 0;   // Initialize the starting index of the longest
palindrome
```

```
    // All substrings of length 1 are palindromes
    for (int i = 0; i < len; i++) {
        table[i][i] = 1;
    }

    // Check for palindromes of length 2
    for (int i = 0; i < len - 1; i++) {
        if (s[i] == s[i + 1]) {
            table[i][i + 1] = 1;
            max_len = 2;
            start = i;
        }
    }

    // Check for palindromes of length greater than 2
    for (int curr_len = 3; curr_len <= len; curr_len++) {
        for (int i = 0; i < len - curr_len + 1; i++) {
            int j = i + curr_len - 1; // Ending index of the current
substring

            // Check if the characters at the current indices match and if
the
            // substring between them is a palindrome
            if (s[i] == s[j] && table[i + 1][j - 1] == 1) {
                table[i][j] = 1;

                // Update the maximum length and starting index if needed
                if (curr_len > max_len) {
                    max_len = curr_len;
                    start = i;
                }
            }
        }
    }

    // Extract and return the longest palindrome substring
    char* result = malloc(max_len + 1);
    strncpy(result, s + start, max_len);
    result[max_len] = '\0';
```

```
    return result;
}

int main() {
    char input_str[] = "babad";
    char* result = longestPalindrome(input_str);

    printf("Longest palindrome substring: %s\n", result);

    // Free dynamically allocated memory
    free(result);

    return 0;
}
```

5) Check if the string contains only digits.

WTD: Determine if all characters in the provided string are numeric digits.
(e.g.: I/P: "1234a" ,O/P: "False")

```
#include <stdio.h>
#include <string.h>

int isNumeric(const char *str) {
    int len = strlen(str);

    // Iterate through the characters in the string
    for (int i = 0; i < len; i++) {
        // Check if the current character is not a digit
        if (str[i] < '0' || str[i] > '9') {
            return 0; // Not a numeric digit
        }
    }

    return 1; // All characters are numeric digits
}
```

```
int main() {
    const char *input_str = "1234a";

    if (isNumeric(input_str)) {
        printf("True\n");
    } else {
        printf("False\n");
    }

    return 0;
}
```

6) Duplicate characters found in a string.

WTD: Spot all characters in the string that appear more than once and list them.
(e.g.: I/P: "programming" ,O/P: "r","g","m")

```
#include <stdio.h>
#include <string.h>

void findDuplicateCharacters(const char *str) {
    int char_count[256] = {0}; // Initialize an array to store character
counts (assuming ASCII characters)

    int len = strlen(str);

    // Iterate through the string and count the occurrences of each
character
    for (int i = 0; i < len; i++) {
        char_count[(int)str[i]]++;
    }

    // Iterate through the character counts and print duplicate characters
    printf("Duplicate characters: ");
    for (int i = 0; i < 256; i++) {
        if (char_count[i] > 1) {
            printf("%c,", (char)i); // Print characters with count > 1
        }
```

```
    }
    printf("\n");
}

int main() {
    const char *input_str = "programming";
    findDuplicateCharacters(input_str);
    return 0;
}
```

7) Check if a string has balanced parentheses.

WTD: Ensure that for every opening bracket, parenthesis, or brace in the string, there's a corresponding closing counterpart, and they are correctly nested.
(e.g.: I/P: "()[{}]" ,O/P: "True")

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Structure to represent a stack
struct Stack {
    int top;
    char *array;
};

// Function to create a new stack
struct Stack* createStack(int size) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->top = -1;
    stack->array = (char*)malloc(size * sizeof(char));
    return stack;
}

// Function to check if the stack is empty
bool isEmpty(struct Stack* stack) {
    return (stack->top == -1);
}
```

```
// Function to push a character onto the stack
void push(struct Stack* stack, char item) {
    stack->array[++stack->top] = item;
}

// Function to pop a character from the stack
char pop(struct Stack* stack) {
    if (!isEmpty(stack)) {
        return stack->array[stack->top--];
    }
    return '\0'; // Return null character if the stack is empty
}

// Function to check if the string has balanced parentheses, brackets, and
braces
bool isBalanced(char* str) {
    int len = strlen(str);
    struct Stack* stack = createStack(len);

    for (int i = 0; i < len; i++) {
        if (str[i] == '(' || str[i] == '[' || str[i] == '{') {
            push(stack, str[i]);
        } else if (str[i] == ')' || str[i] == ']' || str[i] == '}') {
            if (isEmpty(stack)) {
                return false; // Closing symbol with no corresponding
opening symbol
            }

            char top = pop(stack);

            // Check if the current closing symbol matches the top of the
stack
            if ((str[i] == ')' && top != '(') ||
                (str[i] == ']' && top != '[') ||
                (str[i] == '}' && top != '{')) {
                return false; // Mismatched symbols
            }
        }
    }
```

```
    return isEmpty(stack); // Stack should be empty for balanced symbols
}

int main() {
    char input_str[] = "()[{}]";
    bool result = isBalanced(input_str);

    if (result) {
        printf("True\n");
    } else {
        printf("False\n");
    }

    return 0;
}
```

8) Count the occurrences of a given character in a string.

WTD: Count how many times a specified character appears in a given string.
(e.g.: I/P: "apple",Char: 'p' ,O/P: "2")

```
#include <stdio.h>
#include <string.h>

int countCharacterOccurrences(const char *str, char target) {
    int count = 0;
    int len = strlen(str);

    // Iterate through the string and count occurrences of the target
character
    for (int i = 0; i < len; i++) {
        if (str[i] == target) {
            count++;
        }
    }

    return count;
```

```
}

int main() {
    const char *input_str = "apple";
    char target_char = 'p';
    int result = countCharacterOccurrences(input_str, target_char);

    printf("Occurrences of '%c': %d\n", target_char, result);

    return 0;
}
```

9) Check if two strings are anagrams of each other.

WTD: Determine if two provided strings can be rearranged to form each other, meaning they are anagrams.
(e.g.: I/P: "listen" ,O/P: "silent")

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Function to check if two strings are anagrams
bool areAnagrams(const char *str1, const char *str2) {
    int count1[26] = {0}; // Initialize an array to store character counts
for str1 (assuming lowercase letters only)
    int count2[26] = {0}; // Initialize an array to store character counts
for str2

    int len1 = strlen(str1);
    int len2 = strlen(str2);

    // If the lengths of the strings are different, they cannot be
anagrams
    if (len1 != len2) {
        return false;
    }
```

```
    // Count characters in str1
    for (int i = 0; i < len1; i++) {
        count1[str1[i] - 'a']++;
    }

    // Count characters in str2
    for (int i = 0; i < len2; i++) {
        count2[str2[i] - 'a']++;
    }

    // Compare the character counts
    for (int i = 0; i < 26; i++) {
        if (count1[i] != count2[i]) {
            return false; // If any character count differs, they are not
anagrams
        }
    }

    return true; // If all character counts are the same, they are
anagrams
}

int main() {
    const char *str1 = "listen";
    const char *str2 = "silent";

    if (areAnagrams(str1, str2)) {
        printf("The strings are anagrams.\n");
    } else {
        printf("The strings are not anagrams.\n");
    }

    return 0;
}
```

10) Reverse words in a given sentence without using any library method.

WTD: Invert the order of words in a given sentence, maintaining the order of characters within each word.
(e.g.: I/P: "Hello Word" ,O/P: "World Hello")

```c
#include <stdio.h>
#include <string.h>

// Function to reverse a string from start to end
void reverseString(char *start, char *end) {
    while (start < end) {
        char temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}

// Function to reverse words in a sentence
void reverseWords(char *sentence) {
    int len = strlen(sentence);

    // Reverse the entire sentence first
    reverseString(sentence, sentence + len - 1);

    // Initialize pointers for word reversal
    char *word_start = sentence;
    char *word_end = sentence;

    // Iterate through the sentence
    while (*word_end != '\0') {
        if (*word_end == ' ') {
            // Found a space, reverse the current word
            reverseString(word_start, word_end - 1);

            // Move to the next word (skip spaces)
```

```
            while (*word_end == ' ') {
                word_end++;
            }

            // Update word_start to the beginning of the next word
            word_start = word_end;
        } else {
            word_end++;
        }
    }

    // Reverse the last word (or the only word if there's only one)
    reverseString(word_start, word_end - 1);
}

int main() {
    char input_sentence[] = "Hello Word";
    reverseWords(input_sentence);
    printf("Reversed sentence: %s\n", input_sentence);

    return 0;
}
```

11) Check if two strings are a rotation of each other.

WTD: Verify if one string can be obtained by rotating another string, indicating how many positions were involved in the rotation.
(e.g.: I/P: "abcde" "cdeab" ,O/P: "Rotation: 2L"(Obtaining String B by rotating String A))

```
#include <stdio.h>
#include <string.h>

// Function to check if one string is a rotation of another and determine
the rotation position
int isRotation(const char *str1, const char *str2) {
    int len1 = strlen(str1);
    int len2 = strlen(str2);
```

```
    // Check if both strings have the same length and are not empty
    if (len1 != len2 || len1 == 0) {
        return -1; // Invalid or not a rotation
    }

    // Concatenate str1 with itself
    char concatenatedStr[2 * len1 + 1]; // +1 for the null terminator
    strcpy(concatenatedStr, str1);
    strcat(concatenatedStr, str1);

    // Check if str2 is a substring of concatenatedStr
    char *rotationPosition = strstr(concatenatedStr, str2);

    if (rotationPosition != NULL) {
        // Calculate the rotation position (index at which str2 starts in
concatenatedStr)
        int rotationIndex = rotationPosition - concatenatedStr;

        if (rotationIndex < len1) {
            // Rotation to the left
            return len1 - rotationIndex;
        } else {
            // Rotation to the right
            return 2 * len1 - rotationIndex;
        }
    } else {
        return -1; // Not a rotation
    }
}

int main() {
    const char *str1 = "abcde";
    const char *str2 = "cdeab";
    int rotation = isRotation(str1, str2);

    if (rotation != -1) {
        printf("Rotation: %d%s\n", rotation, (rotation > 0) ? "L" : "R");
    } else {
        printf("Not a rotation.\n");
    }
```

```
    return 0;
}
```

12) Check if a given string is a palindrome.

WTD: Ascertain if the provided string reads the same forwards and backwards.
(e.g.: I/P: "radar" ,O/P: "True")

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

// Function to check if a string is a palindrome
bool isPalindrome(const char *str) {
    int len = strlen(str);
    int i = 0;        // Start from the beginning of the string
    int j = len - 1;  // Start from the end of the string

    while (i < j) {
        // Ignore non-alphanumeric characters from the beginning
        while (!isalnum(str[i]) && i < j) {
            i++;
        }

        // Ignore non-alphanumeric characters from the end
        while (!isalnum(str[j]) && i < j) {
            j--;
        }

        // Compare the characters (ignoring case)
        if (tolower(str[i]) != tolower(str[j])) {
            return false; // Mismatch found
        }

        i++;  // Move to the next character from the beginning
        j--;  // Move to the next character from the end
```

```
    }

    return true; // No mismatches found, it's a palindrome
}

int main() {
    const char *input_str = "radar";

    if (isPalindrome(input_str)) {
        printf("True\n");
    } else {
        printf("False\n");
    }

    return 0;
}
```

13) Count the number of vowels and constants in a given string.

WTD: Tally the number of vowel and consonant characters in the given string.
(e.g.: I/P: "apple" ,O/P: Vowels: 2, Consonants: 3)

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// Function to count vowels and consonants in a string
void countVowelsAndConsonants(const char *str) {
    int vowels = 0;
    int consonants = 0;
    int len = strlen(str);

    for (int i = 0; i < len; i++) {
        char ch = tolower(str[i]); // Convert the character to lowercase
for case insensitivity
```

```
        if (ch >= 'a' && ch <= 'z') { // Check if the character is an
alphabet
            if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch ==
'u') {
                vowels++; // It's a vowel
            } else {
                consonants++; // It's a consonant
            }
        }
    }

    printf("Vowels: %d, Consonants: %d\n", vowels, consonants);
}

int main() {
    const char *input_str = "apple";
    countVowelsAndConsonants(input_str);

    return 0;
}
```

14) Reverse a string using recursion.

WTD: Use a recursive method to invert the order of characters in a string.
(e.g.: I/P: "hello" ,O/P: "olleh")

```
#include <stdio.h>
#include <string.h>

// Function to reverse a string using recursion
void reverseStringRecursive(char *str, int start, int end) {
    if (start >= end) {
        return; // Base case: when the start index is greater than or
equal to the end index
    }

    // Swap the characters at the start and end positions
    char temp = str[start];
```

```
    str[start] = str[end];
    str[end] = temp;

    // Recursively reverse the substring between start+1 and end-1
    reverseStringRecursive(str, start + 1, end - 1);
}

int main() {
    char input_str[] = "hello";
    int len = strlen(input_str);

    // Reverse the string using recursion
    reverseStringRecursive(input_str, 0, len - 1);

    printf("Reversed string: %s\n", input_str);

    return 0;
}
```

15) Find all permutations of a string.

WTD: Generate all possible arrangements of the characters from the given string.
(e.g.: I/P: "ab" ,O/P: "ab","ba")

```
#include <stdio.h>
#include <string.h>

// Function to swap two characters in a string
void swap(char *x, char *y) {
    char temp = *x;
    *x = *y;
    *y = temp;
}

// Function to find all permutations of a string
void findPermutations(char *str, int start, int end) {
    if (start == end) {
        printf("%s\n", str); // Print the current permutation
```

```
    } else {
        for (int i = start; i <= end; i++) {
            // Swap characters at positions start and i
            swap(&str[start], &str[i]);

            // Recursively generate permutations for the substring
str[start+1:end]
            findPermutations(str, start + 1, end);

            // Restore the original order of characters for backtracking
            swap(&str[start], &str[i]);
        }
    }
}

int main() {
    char input_str[] = "ab";
    int len = strlen(input_str);

    printf("Permutations of \"%s\":\n", input_str);
    findPermutations(input_str, 0, len - 1);

    return 0;
}
```

16) Check for Pangram

WTD: Determine if the given string contains every letter of the alphabet at least once. For instance, "The quick brown fox jumps over a lazy dog" is a pangram.
(e.g.: I/P: "Jinxed wizards pluck ivy from the big quilt"; O/P: True)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

// Function to check if a string is a pangram
bool isPangram(const char *str) {
```

```
    // Initialize an array to track the occurrence of each letter in the
alphabet
    bool alphabet[26] = {false};

    int len = strlen(str);

    for (int i = 0; i < len; i++) {
        char ch = tolower(str[i]); // Convert the character to lowercase
for case insensitivity

        if (ch >= 'a' && ch <= 'z') { // Check if the character is a
lowercase alphabet
            alphabet[ch - 'a'] = true; // Mark the presence of the letter
        }
    }

    // Check if all letters are present in the alphabet array
    for (int i = 0; i < 26; i++) {
        if (!alphabet[i]) {
            return false; // If any letter is missing, it's not a pangram
        }
    }

    return true; // All letters are present, it's a pangram
}

int main() {
    const char *input_str = "Jinxed wizards pluck ivy from the big quilt";

    if (isPangram(input_str)) {
        printf("True\n");
    } else {
        printf("False\n");
    }

    return 0;
}
```

17) String Interleaving:

WTD: Determine if a given string is an interleaving of two other strings. For example, "abc" and "123" can be interleaved as "a1b2c3".
(e.g.: I/P: Strings: "xyz", "789", Interleaved: "x7y8z9"; O/P: True)

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Function to check if a string is an interleaving of two other strings
bool isInterleaving(const char *str1, const char *str2, const char
*interleaved) {
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int len3 = strlen(interleaved);

    // If the combined length of str1 and str2 is not equal to the length
of interleaved, it's not possible
    if (len1 + len2 != len3) {
        return false;
    }

    // Create a 2D array to store whether substrings of interleaved can be
formed by interleaving str1 and str2
    bool dp[len1 + 1][len2 + 1];
    memset(dp, false, sizeof(dp)); // Initialize the array to false

    // Initialize the base case
    dp[0][0] = true;

    // Fill in the first row
    for (int j = 1; j <= len2; j++) {
        if (str2[j - 1] == interleaved[j - 1] && dp[0][j - 1]) {
            dp[0][j] = true;
        }
    }

    // Fill in the first column
    for (int i = 1; i <= len1; i++) {
        if (str1[i - 1] == interleaved[i - 1] && dp[i - 1][0]) {
```

```
            dp[i][0] = true;
        }
    }

    // Fill in the rest of the table
    for (int i = 1; i <= len1; i++) {
        for (int j = 1; j <= len2; j++) {
            if ((str1[i - 1] == interleaved[i + j - 1] && dp[i - 1][j]) ||
(str2[j - 1] == interleaved[i + j - 1] && dp[i][j - 1])) {
                dp[i][j] = true;
            }
        }
    }

    // Return the result at the bottom-right corner of the table
    return dp[len1][len2];
}

int main() {
    const char *str1 = "xyz";
    const char *str2 = "789";
    const char *interleaved = "x7y8z9";

    if (isInterleaving(str1, str2, interleaved)) {
        printf("True\n");
    } else {
        printf("False\n");
    }

    return 0;
}
```

18) Longest Substring Without Repeating Characters:

WTD: Find the length of the longest substring without repeating characters. For "abcabcbb", the answer would be 3, because the longest substring without repeating letters is "abc".

(e.g.: I/P: "pwwkew"; O/P: 3)

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Function to find the length of the longest substring without repeating characters
int lengthOfLongestSubstring(const char *str) {
    int len = strlen(str);

    if (len == 0) {
        return 0; // If the string is empty, the longest substring is of length 0
    }

    int maxLen = 0; // Maximum length of the substring
    int start = 0; // Start of the current substring
    int charIndex[256]; // Hash table to store the last seen index of each character
    memset(charIndex, -1, sizeof(charIndex)); // Initialize all indices to -1

    for (int end = 0; end < len; end++) {
        char ch = str[end];

        // If the character has been seen in the current substring, update the start position
        if (charIndex[ch] >= start) {
            start = charIndex[ch] + 1;
        }

        // Update the last seen index of the character
        charIndex[ch] = end;

        // Calculate the length of the current substring
        int currentLen = end - start + 1;

        // Update the maximum length if needed
        if (currentLen > maxLen) {
```

```
            maxLen = currentLen;
        }
    }

    return maxLen;
}

int main() {
    const char *input_str = "pwwkew";
    int length = lengthOfLongestSubstring(input_str);

    printf("Length of the longest substring without repeating characters:
%d\n", length);

    return 0;
}
```

19) Count Substrings with Equal 0s and 1s.

WTD: Given a binary string, count the number of substrings that have an equal number of 0s and 1s. For "0011", the answer is 4: "00", "11", "0011", and "01".
(e.g.: I/P: "0101"; O/P: 4)

```
#include <stdio.h>
#include <string.h>

// Function to count substrings with equal 0s and 1s
int countEqualSubstrings(const char *str) {
    int len = strlen(str);
    int count = 0; // Initialize the count of equal substrings
    int sum = 0;   // Initialize the prefix sum

    // Create an array to store the count of 0s and 1s encountered
    int countArray[len + 1];
    memset(countArray, 0, sizeof(countArray)); // Initialize to 0

    // Traverse the binary string
    for (int i = 0; i < len; i++) {
        // Convert '0' to -1 and '1' to 1
```

```
        int val = (str[i] == '0') ? -1 : 1;
        sum += val; // Update the prefix sum

        // If the prefix sum is 0, it means there are an equal number of
0s and 1s from the start
        if (sum == 0) {
            count++;
        }

        // If we have seen this prefix sum before, it means there are
equal 0s and 1s in between
        count += countArray[sum + len];
        countArray[sum + len]++; // Increment the count of this prefix sum
    }

    return count;
}

int main() {
    const char *input_str = "0101";
    int count = countEqualSubstrings(input_str);

    printf("Number of substrings with equal 0s and 1s: %d\n", count);

    return 0;
}
```

20) Find Lexicographically Minimal Rotation.

WTD: Determine the lexicographically smallest rotation of a string. For "bca", the rotations are "bca", "cab", and "abc", and the smallest is "abc".
(e.g.: I/P: "dacb"; O/P: "acbd")

```
#include <stdio.h>
#include <string.h>
```

```c
// Function to find the lexicographically minimal rotation of a string
void findLexicographicallyMinimalRotation(const char *str) {
    int len = strlen(str);

    // Create a double-length string by concatenating the original string
to itself
    char doubleStr[2 * len + 1];
    strcpy(doubleStr, str);
    strcat(doubleStr, str);

    // Initialize the lexicographically smallest string
    char minRotation[len + 1];
    strcpy(minRotation, str);

    // Iterate through all possible rotations
    for (int i = 1; i < len; i++) {
        // Compare the current rotation to the lexicographically smallest
        if (strncmp(doubleStr + i, minRotation, len) < 0) {
            strncpy(minRotation, doubleStr + i, len);
            minRotation[len] = '\0';
        }
    }

    printf("Lexicographically minimal rotation: %s\n", minRotation);
}

int main() {
    const char *input_str = "dacb";
    findLexicographicallyMinimalRotation(input_str);

    return 0;
}
```

21) Substring Count.

WTD: Find out how many times a particular substring appears in the given string.
(e.g.: I/P: Main String: "banana", Substring: "ana"; O/P:)

```c
#include <stdio.h>
#include <string.h>

// Function to count the number of times a substring appears in a string
int countSubstringOccurrences(const char *mainStr, const char *subStr) {
    int mainLen = strlen(mainStr);
    int subLen = strlen(subStr);
    int count = 0;

    // Iterate through the main string
    for (int i = 0; i <= mainLen - subLen; i++) {
        // Check if the substring matches the portion of the main string
starting at position i
        if (strncmp(mainStr + i, subStr, subLen) == 0) {
            count++;
        }
    }

    return count;
}

int main() {
    const char *mainStr = "banana";
    const char *subStr = "ana";
    int occurrences = countSubstringOccurrences(mainStr, subStr);

    printf("Substring \"%s\" appears %d times in \"%s\"\n", subStr,
occurrences, mainStr);

    return 0;
}
```

22) String Encoding.

WTD: Encode a string by replacing each character with the third character after it in the alphabet. Wrap around to the start of the alphabet after 'z'.

(e.g.: I/P: "abc"; O/P: "def".)

```c
#include <stdio.h>
#include <string.h>

// Function to encode a string
void encodeString(char *str) {
    int len = strlen(str);

    for (int i = 0; i < len; i++) {
        char ch = str[i];

        if (ch >= 'a' && ch <= 'z') {
            // Apply the encoding transformation, wrapping around 'z'
            ch = 'a' + (ch - 'a' + 3) % 26;
        } else if (ch >= 'A' && ch <= 'Z') {
            // Apply the encoding transformation, wrapping around 'Z'
            ch = 'A' + (ch - 'A' + 3) % 26;
        }

        str[i] = ch; // Update the character in the string
    }
}

int main() {
    char input_str[] = "abc";

    printf("Original string: %s\n", input_str);

    encodeString(input_str);

    printf("Encoded string: %s\n", input_str);

    return 0;
}
```

23) String Decoding.

WTD: decode a string by replacing each character with the third character after it in the alphabet. Wrap around to the start of the alphabet after 'z'.
(e.g.: I/P: "def"; O/P: "abc".)

```c
#include <stdio.h>
#include <string.h>

// Function to decode a string
void decodeString(char *str) {
    int len = strlen(str);

    for (int i = 0; i < len; i++) {
        char ch = str[i];

        if (ch >= 'a' && ch <= 'z') {
            // Apply the decoding transformation, wrapping around 'a'
            ch = 'a' + (ch - 'a' - 3 + 26) % 26;
        } else if (ch >= 'A' && ch <= 'Z') {
            // Apply the decoding transformation, wrapping around 'A'
            ch = 'A' + (ch - 'A' - 3 + 26) % 26;
        }

        str[i] = ch; // Update the character in the string
    }
}

int main() {
    char input_str[] = "def";

    printf("Encoded string: %s\n", input_str);

    decodeString(input_str);

    printf("Decoded string: %s\n", input_str);

    return 0;
}
```

24) Maximum Occurring Character.

WTD: Determine which character in a string appears the most times.
(e.g.: I/P: "success"; O/P: "s".)

```c
#include <stdio.h>
#include <string.h>

// Function to find the maximum occurring character in a string
char findMaxOccurringCharacter(const char *str) {
    int len = strlen(str);

    // Create an array to store the frequency count of each character
    int frequency[256] = {0};

    // Initialize variables to keep track of the maximum frequency and the
character with that frequency
    int maxFrequency = 0;
    char maxChar = '\0';

    // Iterate through the string and update the frequency count
    for (int i = 0; i < len; i++) {
        char ch = str[i];
        frequency[ch]++;

        // Check if the current character has a higher frequency than the
previous maximum
        if (frequency[ch] > maxFrequency) {
            maxFrequency = frequency[ch];
            maxChar = ch;
        }
    }

    return maxChar;
}

int main() {
    const char *input_str = "success";
```

```
    char maxChar = findMaxOccurringCharacter(input_str);

    printf("Maximum occurring character: %c\n", maxChar);

    return 0;
}
```

25) Palindrome Partitioning.

WTD: Partition a string such that every substring of the partition is a palindrome. Return the minimum cuts needed.
(e.g.: I/P: "aab"; O/P: 1 (The string can be split as ["aa","b"]).)

```
#include <stdio.h>
#include <string.h>

// Function to check if a string is a palindrome
int isPalindrome(const char *str, int start, int end) {
    while (start < end) {
        if (str[start] != str[end]) {
            return 0; // Not a palindrome
        }
        start++;
        end--;
    }
    return 1; // Palindrome
}

// Function to find the minimum cuts for palindrome partitioning
int minPalindromePartition(const char *str) {
    int len = strlen(str);

    // Create a 2D array to store whether substrings are palindromes
    int isPalindromeArray[len][len];

    // Initialize the array
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
```

```
            isPalindromeArray[i][j] = 0;
        }
    }

    // Initialize the cut array to store minimum cuts
    int minCuts[len];

    // Initialize the cut array with the maximum possible cuts
    for (int i = 0; i < len; i++) {
        minCuts[i] = i;
    }

    for (int end = 1; end < len; end++) {
        for (int start = end; start >= 0; start--) {
            if (str[start] == str[end] && (end - start < 2 ||
isPalindromeArray[start + 1][end - 1])) {
                isPalindromeArray[start][end] = 1;

                if (start > 0) {
                    // Update the minimum cuts if necessary
                    minCuts[end] = (start > 0) ? (minCuts[start - 1] + 1)
: 0;
                } else {
                    minCuts[end] = 0;
                }
            }
        }
    }

    return minCuts[len - 1];
}

int main() {
    const char *input_str = "aab";
    int minCuts = minPalindromePartition(input_str);

    printf("Minimum cuts for palindrome partitioning: %d\n", minCuts);

    return 0;
}
```

26) Repeated Substring Pattern.

WTD: Determine if a given string can be constructed by taking a substring of it and appending multiple copies of the substring together.
(e.g.: I/P: "abab"; O/P: True (Because "ab" is repeated))

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Function to check if a string can be formed by repeating a substring
bool repeatedSubstringPattern(const char *str) {
    int len = strlen(str);

    // Iterate through possible substring lengths
    for (int substringLen = 1; substringLen <= len / 2; substringLen++) {
        if (len % substringLen == 0) {
            int repetitions = len / substringLen;
            bool isRepeated = true;

            // Check if the string can be formed by repeating the current substring
            for (int i = 0; i < len; i++) {
                if (str[i] != str[i % substringLen]) {
                    isRepeated = false;
                    break;
                }
            }

            if (isRepeated) {
                return true;
            }
        }
    }
```

```
    return false;
}

int main() {
    const char *input_str = "abab";
    bool result = repeatedSubstringPattern(input_str);

    if (result) {
        printf("The string can be formed by repeating a substring.\n");
    } else {
        printf("The string cannot be formed by repeating a substring.\n");
    }

    return 0;
}
```