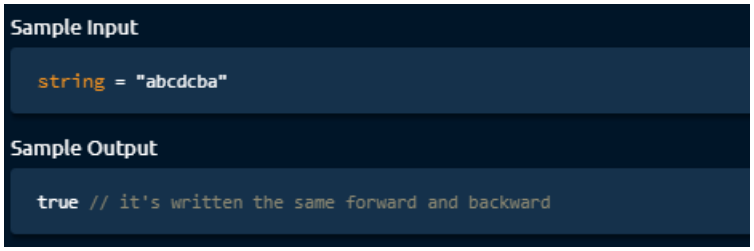
Topic - Strings

Difficulty Easy: Palindrome Check

Write a function that takes in a non-empty string and that returns a Boolean representing whether the string is a palindrome. A palindrome is defined as a string that's written the same forward and backward. Note that single-character strings are palindromes.



Hint:

Start by building the input string in reverse order and comparing this newly built string to the input string. Can you do this without using string concatenations? Can you optimize your algorithm by using recursion? What are the implications of recursion on an algorithm's space-time complexity analysis? Go back to an iterative solution and try using pointers to solve this problem: start with a pointer at the first index of the string and a pointer at the final index of the string. What can you do from there?

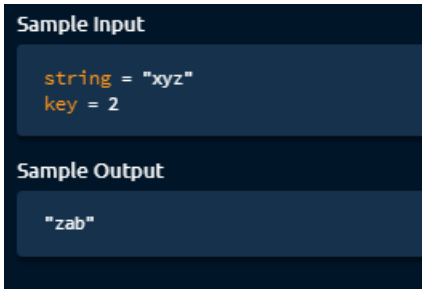
Space-TIme Complexity:

O(n) time | O(1) space - where n is the length of the input string

```python
def isPalindrome(string):
    leftIdx = 0
    rightIdx = len(string) - 1
    while leftIdx < rightIdx:
        if string[leftIdx] != string[rightIdx]:
            return False
        leftIdx += 1
        rightIdx -= 1
    return True
```

Difficulty Easy: Caesar Cipher Encryptor

Given a non-empty string of lowercase letters and a non-negative integer representing a key, write a function that returns a new string obtained by shifting every letter in the input string by k positions in the alphabet, where k is the key. Note that letters should "wrap" around the alphabet; in other words, the letter z shifted by one returns the letter a



Hint:

Most languages have built-in functions that give you the Unicode value of a character as well as the character corresponding to a Unicode value. Consider using such functions to determine which letters the input string's letters should be mapped to. Try creating your own mapping of letters to codes. In other words, try associating each letter in the alphabet with a specific number - its position in the alphabet, for instance - and using that to determine which letters the input string's letters should be mapped to. How do you handle cases where a letter gets shifted to a position that requires wrapping around the alphabet? What about cases where the key is very large and causes multiple wrappings around the alphabet? The modulo operator should be your friend here.

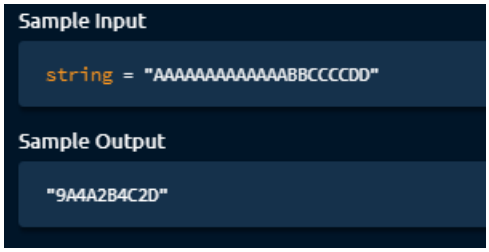
Space-Time Complexity:

O(n) time | O(n) space - where n is the length of the input string

```
def caesarCipherEncryptor(string, key):
    newLetters = []
    newKey = key % 26
    alphabet = list("abcdefghijklmnopqrstuvwxyz")
    for letter in string:
        newLetters.append(getNewLetter(letter, newKey, alphabet))
    return "".join(newLetters)


def getNewLetter(letter, key, alphabet):
    newLetterCode = alphabet.index(letter) + key
    return alphabet[newLetterCode % 26]
```

Difficulty Easy: Run-Length Encoding

Write a function that takes in a non-empty string and returns its run-length encoding. From Wikipedia, "run-length encoding is a form of lossless data compression in which runs of data are stored as a single data value and count, rather than as the original run." For this problem, a run of data is any sequence of consecutive, identical characters. So, the run "AAA" would be run-length-encoded as "3A". To make things more complicated, however, the input string can contain all sorts of special characters, including numbers. And since encoded data must be decodable, this means that we can't naively run-length-encode long runs. For example, the run "AAAAAAAAAAAA" (12 A’s), can't naively be encoded as "12A", since this string can be decoded as either "AAAAAAAAAAAA" or "1AA". Thus, long runs (runs of 10 or more characters) should be encoded in a split fashion; the aforementioned run should be encoded as "9A3A".



Hint:

Traverse the input string and count the length of each run. As you traverse the string, what should you do when you reach a run of length 9 or the end of a run? When you reach a run of length 9 or the end of a run, store the computed count for the run as well as its character (you'll likely need a list for these computed counts and characters), and reset the count to 1 before continuing to traverse the string. Make sure that your solution correctly handles the last run in the string.

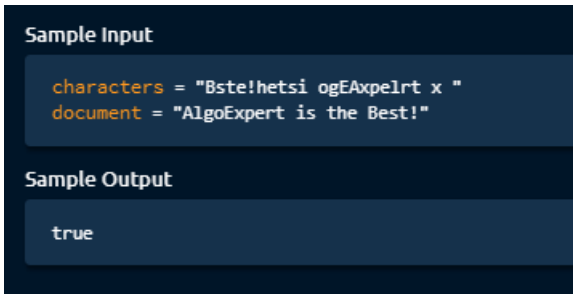
Space-TIme Complexity:

O(n) time | O(n) space - where n is the length of the input string

```python
def runLengthEncoding(string):
    # The input string is guaranteed to be non-empty,
    # so our first run will be of at least length 1.
    encodedStringCharacters = []
    currentRunLength = 1

    for i in range(1, len(string)):
        currentCharacter = string[i]
        previousCharacter = string[i - 1]

        if currentCharacter != previousCharacter or currentRunLength == 9:
            encodedStringCharacters.append(str(currentRunLength))
            encodedStringCharacters.append(previousCharacter)
            currentRunLength = 0

        currentRunLength += 1

    # Handle the last run.
    encodedStringCharacters.append(str(currentRunLength))
    encodedStringCharacters.append(string[len(string) - 1])

    return "".join(encodedStringCharacters)
```

Difficulty Easy: Generate Document

You're given a string of available characters and a string representing a document that you need to generate. Write a function that determines if you can generate the document using the available characters. If you can generate the document, your function should return true; otherwise, it should return false. You're only able to generate the document if the frequency of unique characters in the characters string is greater than or equal to the frequency of unique characters in the document string. For example, if you're given characters = "abcabc" and document = "aabbccc" you cannot generate the document because you're missing one c. The document that you need to create may contain any characters, including special characters, capital letters, numbers, and spaces. Note: you can always generate the empty string ("").



Hint:

There are multiple ways to the solve this problem, but not all approaches have an optimal time complexity. Is there any way to solve this problem in better than O(m * (n + m)) or O(n * (n + m)) time, where n is the length of the characters string and m is the length of the document string? One of the simplest ways to solve this problem is to loop through the document string, one character at a time. At every character, you can count how many times it occurs in the document string and in the characters string. If it occurs more times in the document string than in the characters string, then you cannot generate the document. What is the time complexity of this approach? This approach discussed runs in O(m * (n + m)) time. Can you use some external space to optimize this time complexity? You can solve this problem in O(n + m) time. To do so, you need to use a hash table. Start by counting all of the characters in the

characters string and storing these counts in a hash table. Then, loop through the document string, and check if each character is in the hash table and has a value greater than zero. If a character isn't in the hash table or doesn't have a value greater than zero, then you cannot generate the document. If a character is in the hash table and has a value greater than zero, then decrement its value in the hash table to indicate that you've "used" one of these available characters. If you make it through the entire document string without returning false , then you can generate the document.

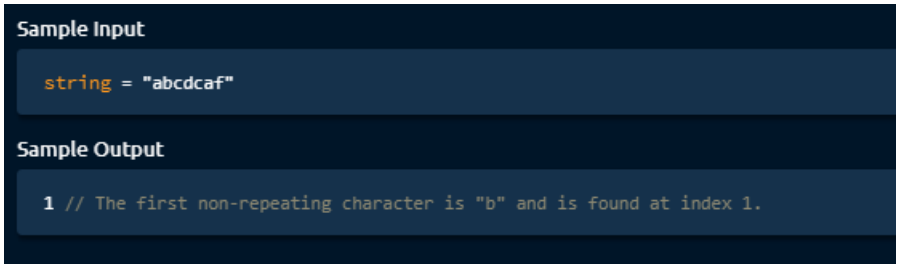
Space-TIme Complexity:

O(n + m) time | O(c) space - where n is the number of characters, m is the length of the document, and c is the number of unique characters in the characters string.

```
def generateDocument(characters, document):
    for character in document:
        documentFrequency = countCharacterFrequency(character, document)
        charactersFrequency = countCharacterFrequency(character, characters)
        if documentFrequency > charactersFrequency:
            return False

    return True


def countCharacterFrequency(character, target):
    frequency = 0
    for char in target:
        if char == character:
            frequency += 1

    return frequency
```

Difficulty Easy: First Non-Repeating Character

Write a function that takes in a string of lowercase English-alphabet letters and returns the index of the string's first non-repeating character. The first non-repeating character is the first character in a string that occurs only once. If the input string doesn't have any non-repeating characters, your function should return -1.



Hint:

How can you determine if a character only appears once in the entire input string? What would be the brute-force approach to solve this problem? One way to solve this problem is with nested traversals of the string: you start by traversing the string, and for each character that you traverse, you traverse through the entire string again to see if the character appears anywhere else. The first index at which you find a character that doesn't appear anywhere else in the string is the index that you return. This approach works, but it's not optimal. Are there any data structures that you can use to improve the time complexity of this approach? Hash tables are very commonly used to keep track of frequencies. Build a hash table, where every key is a character in the string and every value is the corresponding character's frequency in the input string. You can traverse the entire string once to fill the hash table, and then with a second traversal through the string (not a nested traversal), you can use the hash table's constant-time lookups to find the first character with a frequency of 1.

Space-TIme Complexity:

O(n) time | O(1) space - where n is the length of the input string The constant space is because the input string only has lowercase English-alphabet letters; thus, our hash table will never have more than 26 character frequencies.

```python
def firstNonRepeatingCharacter(string):
    for idx in range(len(string)):
        foundDuplicate = False
        for idx2 in range(len(string)):
            if string[idx] == string[idx2] and idx != idx2:
                foundDuplicate = True

        if not foundDuplicate:
            return idx

    return -1
```

Difficulty Medium: Sparse Matrix Multiplication

Write a function that takes in two integer matrices and multiplies them together. Both matrices will be sparse, meaning that most of their elements will be zero. Take advantage of that to reduce the number of total computations that your function performs. If the matrices can't be multiplied together, your function should return [[]].

Sample Input

```
matrix_a = [
  [0,  2, 0],
  [0, -3, 5],
]
matrix_b = [
  [0, 10, 0],
  [0,  0, 0],
  [0,  0, 4],
]
```

Sample Output

```
[
  [0, 0,  0],
  [0, 0, 20],
]
```

Hint:

Matrices A and B can only be multiplied together if the number of columns in matrix A is equal to the number of rows in matrix B. Matrix C, resulting from

multiplying matrices A and B together, has the number of rows that matrix A has and the number of columns that matrix B has. The element at position i, j in matrix C, resulting from multiplying matrices A and B together, is the dot product of the ith row of A and the jth column of B. In other words, c[i, j] = a[i][0] * b[0][j] + a[i][1] * b[1][j] + ... + a[i][k] * b[k][j] . Since a lot of elements in the matrices are zero, you can use sparse representations of the matrices to quickly detect zeroes and to then skip operations that involve multiplications with a zero. This can drastically reduce the total number of operations performed by your function. The sparse representation of a matrix is a dictionary of keys, where each key is a tuple (i, j) and each value is the value found at position i, j in that matrix. If the value at position i, j in the matrix is zero, it isn't included in the dictionary. After transforming the input matrices into their sparse representations, as mentioned in Hint #4, you can check if a particular element is non-zero before performing multiplications with it for various dot products, thereby reducing the number of operations performed by your function.

```python
def sparse_matrix_multiplication(matrix_a, matrix_b):
    if len(matrix_a[0]) != len(matrix_b):
        return [[]]

    sparse_a = get_dict_of_nonzero_cells(matrix_a)
    sparse_b = get_dict_of_nonzero_cells(matrix_b)

    matrix_c = [[0] * len(matrix_b[0]) for _ in range(len(matrix_a))]

    for i, k in sparse_a.keys():
        for j in range(len(matrix_b[0])):
            if (k, j) in sparse_b.keys():
                matrix_c[i][j] += sparse_a[(i, k)] * sparse_b[(k, j)]
    return matrix_c


def get_dict_of_nonzero_cells(matrix):
    dict_of_nonzero_cells = {}
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] != 0:
                dict_of_nonzero_cells[(i, j)] = matrix[i][j]
    return dict_of_nonzero_cells
```