The goal of this exercise is to convert a string to a new string where each character in the new string is "(" if that character appears only once in the original string, or ")" if that character appears more than once in the original string. Ignore capitalization when determining if a character is a duplicate.

Examples

"din" => "((("

"recede" => "()()()"

"Success" => ")())())"

"(( @" => "))(("

function duplicateEncoder(word) {

    const lowerWord = word.toLowerCase();

    const charCount = {}; // this is an object declaration

    for (const char of lowerWord)

    {

        charCount[char] = (charCount[char] || 0) + 1;

    }

    return [...lowerWord].map(char => charCount[char] > 1 ? ')' : '(').join('');

}

console.log(duplicateEncoder('din'));

console.log(duplicateEncoder('recede'));

console.log(duplicateEncoder('Success'));

console.log(duplicateEncoder('(( @'));

To run

> node duplicateEncoder.js

Comparison

function duplicateEncode(word) {

    const lowerCaseWord = word.toLowerCase();

    return lowerCaseWord.split('').map(char =>

        lowerCaseWord.indexOf(char) === lowerCaseWord.lastIndexOf(char) ? '(' : ')'

    ).join('');

}

The difference is, For each character, it checks whether its first occurrence (indexOf) matches its last occurrence (lastIndexOf). If the indices match, the character is unique; otherwise, it is a duplicate. · For each character, the indexOf and lastIndexOf methods scan the string (O(n) each).

· Since this check is performed for every character in the string, the overall complexity is **O(n²)** for strings of length n.

· **Time Complexity**:

* · **My implementation** is more efficient at **O(n)** since it separates counting and transformation.
* **Your implementation** is less efficient at **O(n²)** because it repeatedly scans the string for each character.

· **Space Complexity**:

* · **My implementation** uses extra space to store the charCount object, making it **O(n)** in space usage.
* **Your implementation** does not use additional storage for counts, so its space complexity is **O(1)**.

JAVA

## Java HashMap

In the [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "****key****/****value****" pairs, and you can access them by an index of another type (e.g. a String).

One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

import java.util.HashMap;

public class DuplicateEncoder {

    public static String duplicateEncoder(String word) {

        word = word.toLowerCase();

        HashMap<Character, Integer> charCount = new HashMap<>();

        // this loop populates the hashmap with character counts

        for (char ch : word.toCharArray()) {

            charCount.put(ch, charCount.getOrDefault(ch, 0) + 1); // Retrieves the value associated with the key c. If

                                                                  // the key doesn't exist, it returns the default value

                                                                  // (0).

        }

        StringBuilder result = new StringBuilder(); // A mutable object for building strings efficiently (better than

                                                    // concatenation in loops).

        for (char ch : word.toCharArray()) { // we want to retrieve the count of c and append ( or ) appropriately

            result.append(charCount.get(ch) > 1 ? ')' : '(');

        }

        return result.toString();

    }

    public static void main(String[] args) {

        System.out.println(duplicateEncoder("din"));

        System.out.println(duplicateEncoder("recede"));

        System.out.println(duplicateEncoder("Success"));

        System.out.println(duplicateEncoder("(( @"));

    }

}

PYTHON

def duplicate\_encoder(word):

    word = word.lower()

    char\_count = {char : word.count(char) for char in word} # this is a dictionary where each unique character in the string is a key and the corresponding value is the count. now char is the key and the word.count(char) is the count the for loop iterates through the word

    return ''.join(')' if char\_count[char] > 1 else '(' for char in word) # the join statement joins the statements in the parenthesis into a string

print(duplicate\_encoder('din'))

print(duplicate\_encoder('recede'))

print(duplicate\_encoder('Success'))

print(duplicate\_encoder('(( @'))

C

#include<stdio.h>

#include<string.h>

#include<ctype.h>

void duplicate\_encode(const char \*word, char \*result)

// we have the word variable as a constant so that it cannot be modified within the function . provides read-only access to the string

{

    int char\_count[256] = {0};

    char temp\_word[strlen(word) + 1]; // this is an array

    for (size\_t i = 0; word[i]; i++)

    {

        temp\_word[i] = tolower(word[i]); // convert to a lower case

        char\_count[(unsigned char)temp\_word[i]]++;

    }

    temp\_word[strlen(word)] = '\0';

    for (size\_t i = 0 ; temp\_word[i] ; i++)

    {

        result[i] = char\_count[(unsigned char)temp\_word[i]] > 1 ? ')' : '(';

    }

    result[strlen(word)] = '\0';

}

int main()

{

    const char \*test\_cases[] = {"din", "recede" , "Success", "(( @"};

    char result[256];

    for(int i = 0 ; i < 4 ; i++)

    {

        duplicate\_encode(test\_cases[i], result);

        printf("Input %s -> Output %s\n", test\_cases[i], result);

    }

    return 0;

}

### ****2. Why are**** word ****and**** result ****pointers?****

#### Strings in C:

* Strings in C are arrays of characters. Arrays in C decay into pointers when passed to functions.
  + For example, if you pass "din" to duplicate\_encode, the address of the first character ('d') is passed.

#### Why pointers are needed:

* const char \*word:
  + Since word is an input string, we pass it as a pointer to avoid copying the entire string, which would be inefficient for large strings.
* char \*result:
  + The function needs to modify result to store the output string. Passing it as a pointer allows the function to modify the caller’s buffer directly.

### ****3. Why use**** unsigned char****?****

#### Character Representation:

* Characters in C are stored as char, which can be signed or unsigned depending on the compiler.
  + A **signed char** can represent values from -128 to 127.
  + An **unsigned char** can represent values from 0 to 255.

#### Why it matters:

* **ASCII characters** have values between 0 and 127, but non-ASCII characters can have values greater than 127 or even negative (if char is signed).
* Using unsigned char ensures that all character values are interpreted correctly as non-negative indices for the char\_count array.

### ****4. Why use**** size\_t ****in the loop declaration?****

#### What is size\_t?

* size\_t is an unsigned integer type that is specifically designed to represent the size of objects in memory.
* It is returned by functions like strlen() to ensure compatibility with memory sizes.

#### Why use size\_t?

* When working with string lengths or array indices, it’s common to use size\_t because it guarantees sufficient size for addressing memory, regardless of the platform.
* On a 32-bit system, size\_t is 4 bytes; on a 64-bit system, it is 8 bytes.

#### Example:

c

Copy code

for (size\_t i = 0; word[i]; i++) { /\* do something \*/ }

* Ensures the loop variable i matches the type of values returned by strlen().

C++

#include<iostream>

#include<string>

#include<unordered\_map>

#include<cctype>

using namespace std;

string duplicateEncoder(const string &word)

{

    unordered\_map<char, int> charCount;

    string lowerWord = "";

    for (char ch : word)

    {

        lowerWord += tolower(ch);

    }

    for (char ch : lowerWord)

    {

        charCount[ch]++;

    }

    string result = "";

    for(char ch : lowerWord)

    {

        result += charCount[ch] > 1 ? ')' : '(';

    }

    return result;

}

int main()

{

    string input;

    input = "din";

    cout << "Input: " << input << " -> Output: " << duplicateEncoder(input) << endl;

    input = "recede";

    cout << "Input: " << input << " -> Output: " << duplicateEncoder(input) << endl;

    input = "Success";

    cout << "Input: " << input << " -> Output: " << duplicateEncoder(input) << endl;

    input = "(( @";

    cout << "Input: " << input << " -> Output: " << duplicateEncoder(input) << endl;

    return 0;

}

C#

> dotnet new console

using System;

using System.Linq;

public class Program {

    public static string DuplicateEncode(string word) {

        word = word.ToLower();

        return string.Concat(word.Select(c => word.Count(x => x == c) > 1 ? ')' : '('));

    }

    public static void Main() {

        // Test cases

        Console.WriteLine(DuplicateEncode("din"));

        Console.WriteLine(DuplicateEncode("recede"));

        Console.WriteLine(DuplicateEncode("Success"));

        Console.WriteLine(DuplicateEncode("(( @"));

    }

}

> dotnet run

· word.Select(c => ...):

* · word is a string, and Select is a LINQ method that projects each character in the string into a new sequence.
* The Select method takes a lambda expression c => ..., where c is the parameter representing each character in the string word. It processes each character in word and transforms it into a result based on the logic in the lambda expression.

· word.Count(x => x == c):

* · The Count method counts how many times a specific condition is true in a collection. In this case, it's counting how many times the character c appears in the string word.
* x is a parameter representing each character in the word during the count operation. The condition is x == c, so it's counting the occurrences of c in word.
* For example, if word = "recede" and c = 'e', word.Count(x => x == c) will count how many times the character 'e' appears in the word "recede". The result would be 3.

· word.Count(x => x == c) > 1 ? ')' : '(':

* · This is a **ternary conditional operator**. It checks if the count of the character c in word is greater than 1.
  + If the count is greater than 1 (meaning the character c is a duplicate in the string), it returns ')'.
  + If the count is 1 (meaning the character c appears only once), it returns '('.

· string.Concat(...):

* · The Concat method combines all the characters returned by the Select method into a single string. The result will be the final encoded string, where each character is either ')' or '(' depending on its frequency in the original string.

PHP

<?php

function duplicateEncoder($word)

{

    $word = strtolower($word);

    $charCount = array\_count\_values(str\_split($word));

    $result = "";

    foreach (str\_split($word) as $char)

    {

        $result .= $charCount[$char] > 1 ? ")" : "(" ;

    }

    return $result;

}

echo duplicateEncoder("din") . "\n";

echo duplicateEncoder("recede") . "\n";

echo duplicateEncoder("Success") . "\n";

echo duplicateEncoder("(( @") . "\n";

?>

> php DuplicateEncoder.php