Day-15 SRE Training

Topic: Python DSA

Question: Remove Outer Parentheses

Given a valid parentheses string `s`, remove the outermost parentheses and return the resulting string.

A **primitive valid parentheses** substring is a non-empty substring that is valid and cannot be split into smaller valid parts.

Example 1:

Input:
`s = "(()())(())"`
Output:
`"()()()"`

Example 2:

Input:
`s = "(()())(())(()(()))"`
Output:
`"()()()()(())"`

```
def removeOuterParentheses(s):
    result = []
    open_count = 0  # Tracks open parentheses count

    for char in s:
        if char == '(':
            if open_count > 0:  # Ignore outermost '('
                result.append(char)
            open_count += 1
        else:  # char == ')'
            open_count -= 1
            if open_count > 0:  # Ignore outermost ')'
                result.append(char)

    return "".join(result)
```

```
# Test cases
print(removeOuterParentheses("(()())(())"))  # Output: "()()()"
print(removeOuterParentheses("(()())(())(()(()))"))  # Output:
"()()()()(())"
```

Question: Reverse Words in a String

Given a string `s`, reverse the words in it while maintaining their order.

Example 1:

Input:
`s = "Hello World"`
Output:
`"olleH dlroW"`

Example 2:

Input:
`s = "Python is fun"`
Output:
`"nohtyP si nuf"`

Split the string into words using `split()`, then reverse each word using slicing `[::-1]`. Finally, join the reversed words back into a string using `" ".join()`. The time complexity is **O(N)** as each operation runs linearly.

```
def reverseWords(s):
    words = s.split()  # Step 1: Split the string into words
    reversed_words = [word[::-1] for word in words]  # Step 2: Reverse each
word
    return " ".join(reversed_words)  # Step 3: Join them back

# Test cases
print(reverseWords("Hello World"))  # Output: "olleH dlroW"
print(reverseWords("Python is fun"))  # Output: "nohtyP si nuf"
```

Question: Longest Common Substring in an Array of Strings

Example 1:

Input:
`s1 = "abcde", s2 = "abfce"`
Output:
2 (Common substring: "ab")

Given an array of strings, find the longest common substring present in all strings without using dynamic programming.

1. **Find the shortest string** in the array (since the longest possible substring cannot be longer than it).
2. **Iterate over all substrings** of the shortest string, starting from the longest.
3. **Check if the substring is present in all strings** in the array.
4. **Return the longest valid substring** found.

```
def longestCommonSubstring(arr):
    if not arr:
        return ""

    shortest = min(arr, key=len)  # Step 1: Find the shortest string

    for length in range(len(shortest), 0, -1):  # Step 2: Iterate over possible substrings
        for start in range(len(shortest) - length + 1):
            substring = shortest[start:start + length]
            if all(substring in s for s in arr):  # Step 3: Check if present in all strings
                return substring  # Step 4: Return the longest valid substring

    return ""

# Test cases
print(longestCommonSubstring(["flower", "flow", "flight"]))  # Output: "fl"
print(longestCommonSubstring(["abcd", "bcda", "cdbc"]))  # Output: "bcd"
```

Time Complexity: O(N * L^2), where N is the number of strings and L is the length of the shortest string.

Space Complexity: O(1), since no extra space is used except for variables.

Question: Check if One String is a Rotation of Another

Given two strings, determine if one is a rotation of the other.

Example 1:

Input:
`s1 = "waterbottle", s2 = "erbottlewat"`
Output:
`True` (s2 is a rotation of s1)

Approach: Checking All Possible Rotations

This method checks if one string is a rotation of another by **generating all possible rotations** of `s1` and comparing them with `s2`.

1. **Check Lengths:**
   - If `s1` and `s2` have different lengths, return `False` immediately since a rotation must preserve length.
2. **Iterate Over All Rotations:**
   - Loop through each index `i` of `s1`.
   - Generate a rotated version by splitting `s1` into two parts:
     - `s1[i:]` → From index `i` to end.
     - `s1[:i]` → From start to index `i`.
   - Concatenating these two parts (`s1[i:] + s1[:i]`) produces a rotated version of `s1`.
3. **Compare with `s2`:**
   - If a rotated version matches `s2`, set `flag = True` and break the loop.
4. **Return the Result:**
   - If a valid rotation is found, return `True`, otherwise return `False`.

```
def isRotation(s1, s2):
    if len(s1) != len(s2):
        return False  # Different lengths → Not a rotation

    flag = False  # Initialize flag as False

    # Try all possible rotations
    for i in range(len(s1)):
        rotated = s1[i:] + s1[:i]  # Rotate the string by shifting
characters
        if rotated == s2:
            flag = True  # Found a valid rotation
            break  # No need to check further

    return flag  # Return the flag

# Example usage:
print(isRotation("waterbottle", "erbottlewat"))  # Output: True
print(isRotation("hello", "lohel"))  # Output: True
print(isRotation("hello", "olelh"))  # Output: False
```

Time Complexity: O(N²)

- Each rotation takes O(N) time to create a new string, and we perform this operation N times.

Space Complexity: O(N)

- Each rotated string takes extra space.

Approach: Using Concatenation

- If `s1` and `s2` have different lengths, return `False`.
- Concatenate `s1` with itself (`s1 + s1`).
- Check if `s2` is a substring of this concatenated string.

```
def isRotation(s1, s2):
    if len(s1) != len(s2):
        return False
    return s2 in (s1 + s1)

# Test cases
print(isRotation("waterbottle", "erbottlewat"))  # Output: True
print(isRotation("hello", "lohel"))  # Output: True
print(isRotation("abc", "acb"))  # Output: False
```

Time Complexity: O(N)

Space Complexity: O(N)

Question: Check if Two Strings are Anagrams

Two strings are anagrams if they contain the same characters with the same frequency, but in any order.

Example 1:

Input:
`s1 = "listen", s2 = "silent"`
Output:
`True` (Both have the same characters)

Approach 1: Using Sorting

- Sort both strings and compare them.

```
def isAnagram(s1, s2):
    return sorted(s1) == sorted(s2)

print(isAnagram("listen", "silent"))  # Output: True
print(isAnagram("hello", "world"))    # Output: False
```

- **Time Complexity:** O(N log N)
- **Space Complexity:** O(1)

Approach 2: Using HashMap (Efficient)

- Count character frequencies using `Counter` from `collections`.

```
from collections import Counter

def isAnagram(s1, s2):
    return Counter(s1) == Counter(s2)

print(isAnagram("listen", "silent"))  # Output: True
print(isAnagram("hello", "world"))    # Output: False
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(N)

Using a Character Frequency Array (Optimized for Lowercase Letters)

- Since there are only **26 lowercase English letters**, we can use an array of size 26 instead of a dictionary.

```
def isAnagram(s1, s2):
    if len(s1) != len(s2):
        return False

    freq = [0] * 26  # Array to track character counts

    for c1, c2 in zip(s1, s2):
        freq[ord(c1) - ord('a')] += 1
        freq[ord(c2) - ord('a')] -= 1

    return all(x == 0 for x in freq)

print(isAnagram("listen", "silent"))  # Output: True
print(isAnagram("hello", "world"))    # Output: False
```

- **Time Complexity:** O(N)
- **Space Complexity:** O(1)