ASSIGNMENT 8

1. Given two strings s1 and s2, return *the lowest **ASCII** sum of deleted characters to make two strings equal*.

```
def minimum_ascii_delete_sum(s1, s2):
    m, n = len(s1), len(s2)

    # Create a 2D table to store the lengths of LCS
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Calculate the lengths of LCS
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + ord(s1[i - 1])
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # Calculate the sum of deleted characters
    ascii_sum = sum(ord(char) for char in s1 + s2) - 2 * dp[m][n]

    return ascii_sum
```

2. Given a string s containing only three types of characters: '(', ')' and '*', return true *if* s *is **valid***.

The following rules define a **valid** string:

- Any left parenthesis '(' must have a corresponding right parenthesis ')'.
- Any right parenthesis ')' must have a corresponding left parenthesis '('.
- Left parenthesis '(' must go before the corresponding right parenthesis ')'.
- '*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string "".

```
def check_valid_string(s):
    stack = []
    star_stack = []

    for i, char in enumerate(s):
        if char == '(':
            stack.append(i)
        elif char == '*':
            star_stack.append(i)
        else:  # char == ')'
            if stack:
                stack.pop()
            elif star_stack:
                star_stack.pop()
            else:
                return False

    while stack and star_stack:
        if stack[-1] > star_stack[-1]:
            return False
        stack.pop()
        star_stack.pop()

    return len(stack) == 0
```

3. Given two strings word1 and word2, return *the minimum number of* ***steps*** *required to make* word1 *and* word2 *the same*.

In one **step**, you can delete exactly one character in either string.

```
def min_steps_to_same(word1, word2):
    m, n = len(word1), len(word2)

    # Create a 2D table to store the lengths of LCS
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Calculate the lengths of LCS
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # Calculate the minimum number of steps
    lcs_length = dp[m][n]
    min_steps = m + n - 2 * lcs_length

    return min_steps
```

4. You need to construct a binary tree from a string consisting of parenthesis and integers.

The whole input represents a binary tree. It contains an integer followed by zero, one or two pairs of parenthesis. The integer represents the root's value and a pair of parenthesis contains a child binary tree with the same structure. You always start to construct the **left** child node of the parent first if it exists.

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def construct_binary_tree(s):
    if not s:
        return None

    # Find the first parenthesis
    first_parenthesis = s.find('(')
    if first_parenthesis == -1:

        val = int(s)
        return TreeNode(val)

    val = int(s[:first_parenthesis])

    open_parenthesis = 0
    for i in range(first_parenthesis, len(s)):
        if s[i] == '(':
            open_parenthesis += 1
        elif s[i] == ')':
            open_parenthesis -= 1
            if open_parenthesis == 0:
                closing_parenthesis = i
                break

    left_subtree = construct_binary_tree(s[first_parenthesis + 1: closing_parenthesis])
    right_subtree = construct_binary_tree(s[closing_parenthesis + 2: -1])

    # Create the root node and attach the subtrees
    root = TreeNode(val, left_subtree, right_subtree)

    return root
```

5. Given an array of characters chars, compress it using the following algorithm:

Begin with an empty string s. For each group of **consecutive repeating characters** in chars:

- If the group's length is 1, append the character to s.
- Otherwise, append the character followed by the group's length.

The compressed string s **should not be returned separately**, but instead, be stored **in the input character array chars**. Note that group lengths that are 10 or longer will be split into multiple characters in chars.

After you are done **modifying the input array,** return *the new length of the array*.

You must write an algorithm that uses only constant extra space.

```
def compress(chars):
    n = len(chars)
    write_idx = 0
    count = 1

    for read_idx in range(1, n):
        if chars[read_idx] == chars[read_idx - 1]:
            count += 1
        else:
            chars[write_idx] = chars[read_idx - 1]
            write_idx += 1

            if count > 1:
                count_str = str(count)
                for digit in count_str:
                    chars[write_idx] = digit
                    write_idx += 1

            count = 1

    # Write the last character and its count
    chars[write_idx] = chars[n - 1]
    write_idx += 1

    if count > 1:
        count_str = str(count)
        for digit in count_str:
            chars[write_idx] = digit
            write_idx += 1

    return write_idx
```

6. Given two strings s and p, return *an array of all the start indices of* p*'s anagrams in* s. You may return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

```
def find_anagrams(s, p):
    result = []
    p_freq = Counter(p)
    window_freq = Counter(s[:len(p)])

    if p_freq == window_freq:
        result.append(0)

    for i in range(len(p), len(s)):
        if window_freq[s[i - len(p)]] == 1:
            del window_freq[s[i - len(p)]]
        else:
            window_freq[s[i - len(p)]] -= 1

        window_freq[s[i]] += 1

        if window_freq == p_freq:
            result.append(i - len(p) + 1)

    return result
```

7. Given an encoded string, return its decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].

The test cases are generated so that the length of the output will never exceed 105.

```
def decode_string(s):
    stack = []

    for char in s:
        if char != ']':
            stack.append(char)
        else:
            # Pop characters until '[' is found
            substr = ''
            while stack and stack[-1] != '[':
                substr = stack.pop() + substr

            # Pop '[' from the stack
            stack.pop()

            # Get the repetition count
            count = ''
            while stack and stack[-1].isdigit():
                count = stack.pop() + count

            # Repeat the substring and push it back to the stack
            stack.append(int(count) * substr)

    # Concatenate the characters left in the stack
    return ''.join(stack)
```

8. Given two strings s and goal, return true _if you can swap two letters in_ s _so the result is equal to_ goal*, otherwise, return* false*.*

Swapping letters is defined as taking two indices i and j (0-indexed) such that i != j and swapping the characters at s[i] and s[j].

- For example, swapping at indices 0 and 2 in "abcd" results in "cbad".

```
def buddy_strings(s, goal):
    if len(s) != len(goal):
        return False

    if s == goal:
        # Check if s has duplicate characters
        return len(set(s)) < len(s)

    diff_indices = []
    for i in range(len(s)):
        if s[i] != goal[i]:
            diff_indices.append(i)

    if len(diff_indices) == 2:
        i, j = diff_indices
        return s[i] == goal[j] and s[j] == goal[i]

    return False
```