**DSA – ASSIGNMENT 8**

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

**Example 1:**

**Input:** s1 = "sea", s2 = "eat"

**Output:** 231

**Explanation:** Deleting "s" from "sea" adds the ASCII value of "s" (115) to the sum.

Deleting "t" from "eat" adds 116 to the sum.

At the end, both strings are equal, and 115 + 116 = 231 is the minimum sum possible to achieve this.

**Solution. :-**

* Create a 2D table, dp, with dimensions (m+1) x (n+1), where m and n are the lengths of s1 and s2, respectively.
  + dp[i][j] represents the lowest ASCII sum of deleted characters to make the substrings s1[0...i-1] and s2[0...j-1] equal.
* Initialize the first row and the first column of dp as follows:
  + dp[i][0] = dp[i-1][0] + ASCII value of s1[i-1] (summing up the ASCII values of characters in s1)
  + dp[0][j] = dp[0][j-1] + ASCII value of s2[j-1] (summing up the ASCII values of characters in s2)
* Iterate over the remaining cells of dp (from row 1 and column 1) and fill them using the following rules:
  + If s1[i-1] == s2[j-1], then dp[i][j] = dp[i-1][j-1] (no characters need to be deleted).
  + Otherwise, dp[i][j] = min(dp[i-1][j] + ASCII value of s1[i-1], dp[i][j-1] + ASCII value of s2[j-1]).
    - dp[i-1][j] represents the sum if we delete the character s1[i-1].
    - dp[i][j-1] represents the sum if we delete the character s2[j-1].
* Return dp[m][n], which represents the lowest ASCII sum of deleted characters to make s1 and s2 equal.

**def minimumDeleteSum(s1, s2):**

**m, n = len(s1), len(s2)**

**dp = [[0] \* (n + 1) for \_ in range(m + 1)]**

**# Initialize the first row**

**for i in range(1, m + 1):**

**dp[i][0] = dp[i-1][0] + ord(s1[i-1])**

**# Initialize the first column**

**for j in range(1, n + 1):**

**dp[0][j] = dp[0][j-1] + ord(s2[j-1])**

**# Fill the dp table**

**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]**

**else:**

**dp[i][j] = min(dp[i-1][j] + ord(s1[i-1]), dp[i][j-1] + ord(s2[j-1]))**

**return dp[m][n]**

**s1 = "sea"**

**s2 = "eat"**

**result = minimumDeleteSum(s1, s2)**

**print(result)**

💡 **Question 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 "".

**Example 1:**

**Input:** s = "()"

**Output:**

True

**Solution. :-**

* Initialize an empty stack.
* Iterate through each character, c, in the string s:
  + If c is '(' or '\*', push it onto the stack.
  + If c is ')':
    - If the stack is not empty and the top element is '(', pop the top element from the stack.
    - Otherwise, if the stack is not empty and the top element is '\*', pop the top element from the stack.
    - If neither of the above conditions is satisfied, return False since there is no corresponding left parenthesis for the current ')'.
* After iterating through all the characters in s, check the remaining elements in the stack:
  + If there are any '(' remaining, but no corresponding ')' or '\*', return False.
  + Otherwise, return True.

**def isValid(s):**

**stack = []**

**for c in s:**

**if c == '(' or c == '\*':**

**stack.append(c)**

**elif c == ')':**

**if stack and stack[-1] == '(':**

**stack.pop()**

**elif stack and stack[-1] == '\*':**

**stack.pop()**

**else:**

**return False**

**count = 0**

**while stack:**

**if stack.pop() == '(':**

**count += 1**

**elif count > 0:**

**count -= 1**

**else:**

**return False**

**return True if count == 0 else False**

**s = "()"**

**result = isValid(s)**

**print(result)**

💡 **Question 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.

**Example 1:**

**Input:** word1 = "sea", word2 = "eat"

**Output:** 2

**Explanation:** You need one step to make "sea" to "ea" and another step to make "eat" to "ea".

**Solution. :-**

* Create a 2D table, dp, with dimensions (m+1) x (n+1), where m and n are the lengths of word1 and word2, respectively.
  + dp[i][j] represents the minimum number of steps required to make the substrings word1[0...i-1] and word2[0...j-1] the same.
* Initialize the first row and the first column of dp as follows:
  + dp[i][0] = i (the minimum number of steps to make word1[0...i-1] empty is i).
  + dp[0][j] = j (the minimum number of steps to make word2[0...j-1] empty is j).
* Iterate over the remaining cells of dp (from row 1 and column 1) and fill them using the following rules:
  + If word1[i-1] == word2[j-1], then dp[i][j] = dp[i-1][j-1] (no characters need to be deleted).
  + Otherwise, dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1).
    - dp[i-1][j] represents the minimum steps if we delete the character word1[i-1].
    - dp[i][j-1] represents the minimum steps if we delete the character word2[j-1].
* Return dp[m][n], which represents the minimum number of steps required to make word1 and word2 the same.

**def minDistance(word1, word2):**

**m, n = len(word1), len(word2)**

**dp = [[0] \* (n + 1) for \_ in range(m + 1)]**

**# Initialize the first row**

**for i in range(1, m + 1):**

**dp[i][0] = i**

**# Initialize the first column**

**for j in range(1, n + 1):**

**dp[0][j] = j**

**# Fill the dp table**

**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]**

**else:**

**dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1)**

**return dp[m][n]**

**word1 = "sea"**

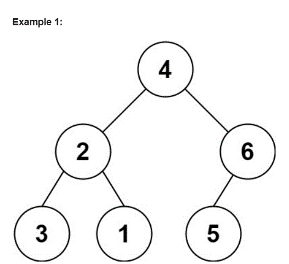
**word2 = "eat"**

**result = minDistance(word1, word2)**

**print(result)**

💡 **Question 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.



**Input:** s = "4(2(3)(1))(6(5))"

**Output:** [4,2,6,3,1,5]

**Solution. :-**

* Define a helper function, constructTree(s, start, end), that takes the input string s and the start and end indices of the current subtree.
* Check if start > end. If true, return None, indicating an empty subtree.
* Find the index of the first opening parenthesis '(' after the start index.
* If no opening parenthesis is found, the substring from start to end represents a single integer. Convert it to an integer value and create a new tree node with that value. Return the node.
* Set the variable parenCount to 0. Iterate through each character, starting from the opening parenthesis index found in step 3:
  + If the character is '(', increment parenCount.
  + If the character is ')', decrement parenCount.
  + If parenCount is 0, it means we have found the matching closing parenthesis for the current subtree.
    - Recursively call the constructTree function to construct the left child subtree, passing the substring between the opening and closing parentheses as the new subtree.
    - Set the index of the matching closing parenthesis as the leftEnd index.
    - Recursively call the constructTree function again to construct the right child subtree, passing the remaining substring after the closing parenthesis as the new subtree.
    - Set the index of the last character of the remaining substring as the rightEnd index.
* Create a new tree node with the integer value found at the start index.
  + Set the left child of the node to the result of the recursive call to constructTree for the left subtree.
  + Set the right child of the node to the result of the recursive call to constructTree for the right subtree.
  + Return the node.
* Call the constructTree function with the input string s and the indices 0 and len(s) - 1 to construct the entire binary tree.

**class TreeNode:**

**def \_\_init\_\_(self, val=0, left=None, right=None):**

**self.val = val**

**self.left = left**

**self.right = right**

**def constructTree(s, start, end):**

**if start > end:**

**return None**

**openingParenIndex = s.find('(', start, end+1)**

**if openingParenIndex == -1:**

**val = int(s[start:end+1])**

**return TreeNode(val)**

**node = TreeNode(int(s[start:openingParenIndex]))**

**parenCount = 0**

**for i in range(openingParenIndex, end+1):**

**if s[i] == '(':**

**parenCount += 1**

**elif s[i] == ')':**

**parenCount -= 1**

**if parenCount == 0:**

**leftEnd = i**

**break**

**rightEnd = end - 1**

**node.left = constructTree(s, openingParenIndex + 1, leftEnd - 1)**

**node.right = constructTree(s, leftEnd + 2, rightEnd)**

**return node**

**def treeFromString(s):**

**return constructTree(s, 0, len(s) - 1)**

**s = "4(2(3)(1))(6(5))"**

**root = treeFromString(s)**

**# Function to traverse and print the tree (for verification)**

**def inorderTraversal(node):**

**if node is None:**

**return**

**inorderTraversal(node.left)**

**print(node.val, end=' ')**

**inorderTraversal(node.right)**

**inorderTraversal(root)**

💡 **Question 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.

**Example 1:**

**Input:** chars = ["a","a","b","b","c","c","c"]

**Output:** Return 6, and the first 6 characters of the input array should be: ["a","2","b","2","c","3"]

**Explanation:**

The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

**Solution. :-**

* Initialize two pointers: write and read, both initially set to 0. The write pointer keeps track of the position where the next compressed character should be written, and the read pointer iterates through the original characters.
* Initialize a variable count to 1 to keep track of the consecutive occurrences of a character.
* Iterate through the array using the read pointer, starting from index 1:
  + If the current character is the same as the previous character, increment count.
  + If the current character is different from the previous character:
    - Write the previous character at the write position in the array.
    - If count is greater than 1 (indicating consecutive occurrences), convert count to a string and write each digit at the subsequent positions in the array, updating write accordingly.
    - Reset count to 1 and update write to the next position.
* After the loop ends, write the last character and its count (if applicable) at the write position.
* Return the value of write, which represents the new length of the compressed array.

**def compress(chars):**

**write = 0**

**read = 0**

**length = len(chars)**

**while read < length:**

**count = 1**

**while read + 1 < length and chars[read] == chars[read + 1]:**

**read += 1**

**count += 1**

**chars[write] = chars[read]**

**write += 1**

**if count > 1:**

**count\_str = str(count)**

**for digit in count\_str:**

**chars[write] = digit**

**write += 1**

**read += 1**

**return write**

**chars = ["a", "a", "b", "b", "c", "c", "c"]**

**result = compress(chars)**

**compressed\_chars = chars[:result]**

**print(result)**

**print(compressed\_chars)**

💡 **Question 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.

**Example 1:**

**Input:** s = "cbaebabacd", p = "abc"

**Output:** [0,6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**Solution. :-**

* Initialize two frequency maps: p\_freq to store the frequencies of characters in string p, and window\_freq to store the frequencies of characters in the current window of string s.
* Initialize two pointers: left and right, both initially set to 0. The left pointer represents the start index of the window, and the right pointer represents the end index of the window.
* Initialize a variable match\_count to 0 to keep track of the number of characters that match between p\_freq and window\_freq.
* Iterate through string s using the right pointer, until it reaches the end of the string:
  + Increment the frequency of the current character at the right index in window\_freq.
  + If the frequency of the current character in window\_freq is equal to or less than the frequency of the same character in p\_freq, increment match\_count.
  + If the window size is greater than the length of string p:
    - Decrement the frequency of the character at the left index in window\_freq.
    - If the frequency of the character at the left index becomes 0, remove it from window\_freq.
    - If the frequency of the character at the left index in window\_freq becomes less than the frequency of the same character in p\_freq, decrement match\_count.
    - Increment the left pointer to slide the window to the right.
  + If match\_count is equal to the number of unique characters in p\_freq, it means all the characters in the current window match the characters in string p.
    - Add the left index to the result array.
* Return the result array.

**from collections import defaultdict**

**def findAnagrams(s, p):**

**p\_freq = defaultdict(int)**

**window\_freq = defaultdict(int)**

**for char in p:**

**p\_freq[char] += 1**

**left = 0**

**match\_count = 0**

**result = []**

**for right in range(len(s)):**

**right\_char = s[right]**

**window\_freq[right\_char] += 1**

**if window\_freq[right\_char] <= p\_freq[right\_char]:**

**match\_count += 1**

**if right - left + 1 > len(p):**

**left\_char = s[left]**

**window\_freq[left\_char] -= 1**

**if window\_freq[left\_char] == 0:**

**del window\_freq[left\_char]**

**if window\_freq[left\_char] < p\_freq[left\_char]:**

**match\_count -= 1**

**left += 1**

**if match\_count == len(p\_freq):**

**result.append(left)**

**return result**

**s = "cbaebabacd"**

**p = "abc"**

**result = findAnagrams(s, p)**

**print(result)**

💡 **Question 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.

**Example 1:**

**Input:** s = "3[a]2[bc]"

**Output:** "aaabcbc"

**Solution. :-**

* Initialize an empty stack.
* Iterate through each character in the input string s:
  + If the current character is a digit, it represents the repetition count.
    - Initialize a variable num as an empty string and keep appending digits to it until a non-digit character is encountered.
    - Convert num to an integer and push it onto the stack.
  + If the current character is an opening bracket '[' or a letter, it represents the start of a substring.
    - Push the current character onto the stack.
  + If the current character is a closing bracket ']':
    - Pop characters from the stack until an opening bracket '[' is encountered.
    - Construct the substring by concatenating the popped characters in reverse order.
    - Pop the repetition count from the stack.
    - Multiply the substring by the repetition count.
    - Push the multiplied substring back onto the stack.
* After iterating through the entire input string, the stack will contain the decoded string.
  + Pop characters from the stack and concatenate them in reverse order to obtain the final decoded string.
* Return the decoded string.

**def decodeString(s):**

**stack = []**

**for char in s:**

**if char == ']':**

**substring = ''**

**while stack[-1] != '[':**

**substring = stack.pop() + substring**

**stack.pop() # Remove '['**

**count = stack.pop()**

**substring = substring \* count**

**stack.append(substring)**

**elif char.isdigit():**

**num = ''**

**while char.isdigit():**

**num += char**

**char = next(s\_iter, None)**

**stack.append(int(num))**

**# Process the character again**

**stack.append(char)**

**else:**

**stack.append(char)**

**decoded\_string = ''**

**while stack:**

**decoded\_string = stack.pop() + decoded\_string**

**return decoded\_string**

**s = "3[a]2[bc]"**

**result = decodeString(s)**

**print(result)**

💡 **Question 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".

**Example 1:**

**Input:** s = "ab", goal = "ba"

**Output:** true

**Explanation:** You can swap s[0] = 'a' and s[1] = 'b' to get "ba", which is equal to goal.

**Solution. :-**

* Initialize two empty lists, diff\_indices and diff\_chars.
* Iterate through the characters in strings s and goal simultaneously, comparing each pair of characters at the same index:
  + If the characters are different, append the index to diff\_indices and the characters to diff\_chars.
* If the length of diff\_indices is 0, it means s and goal are already identical, so return True.
* If the length of diff\_indices is not 2, it means there are more or fewer than two differing characters, so return False.
* If the characters at the differing indices in s and goal are the same but the differing characters themselves are different, return True. This indicates that swapping the differing characters will make the strings equal.
* Otherwise, return False.

**def canBeEqual(s, goal):**

**diff\_indices = []**

**diff\_chars = []**

**for i in range(len(s)):**

**if s[i] != goal[i]:**

**diff\_indices.append(i)**

**diff\_chars.append((s[i], goal[i]))**

**if len(diff\_indices) == 0:**

**return True**

**elif len(diff\_indices) != 2:**

**return False**

**else:**

**char1, char2 = diff\_chars**

**if char1 == char2[::-1]:**

**return True**

**else:**

**return False**

**s = "ab"**

**goal = "ba"**

**result = canBeEqual(s, goal)**

**print(result)**