ASSIGMNENT -8

STRINGS

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.

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
    def minimumDeleteSum(self, s1: str, s2: str) -> int:
        m, n = len(s1), len(s2)
        dp = [[0] * (m + 1) for _ in range(n + 1)]

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

        # ASCII Sum of S1 - Max ASCII Sum of CS +  ASCII Sum of S2 - Max
ASCII Sum of CS
        return sum(map(ord, s1 + s2)) - dp[-1][-1] * 2
```

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

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
    def checkValidString(self, s: str) -> bool:
        leftmin = leftmax = 0
        for c in s:
            if c == "(":
                leftmax += 1
                leftmin += 1
            if c == ")":
                leftmax -= 1
                leftmin = max(0, leftmin-1)
            if c == "*":
                leftmax +=1
                leftmin = max(0, leftmin-1)
            if leftmax < 0:
                return False
        if leftmin == 0:
            return True
```

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".

SOLUTIONS:

TC:O(m*n), SC:O(m*n)

CODE:

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m,n=len(word1),len(word2)
        @cache
        def lcs(i, j): # find longest common subsequence
            if i==m or j==n:
                return 0
            return 1 + lcs(i+1, j+1) if word1[i]==word2[j]
else  max(lcs(i+1, j), lcs(i,j+1))
        # subtract the lcs length from both the strings
        # the difference is the number of characters that has to deleted
        return m + n - 2*lcs(0,0)
```

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.

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
  def str2tree(self, s: str) -> Optional[TreeNode]:
      '''
      have a dfs function to walk through the string
      dfs(s, i), take string and start index
      build node then recursive calls to build node.left and node.right
      return node and index

      returning index because we do preorder traversal
      need next index to build node.right
      '''
      if not s or len(s) == 0:
          return None

      root, idx = self.dfs(s, 0)
```

```
        return root

    def dfs(self, s, i):
        start = i
        if s[start] == '-':
            i += 1
        while i < len(s) and s[i].isdigit():
            i += 1
        node = TreeNode(int(s[start:i]))

        if i < len(s) and s[i] == '(':
            i += 1
            node.left, i = self.dfs(s, i)
            i += 1

        if i < len(s) and s[i] == '(':
            i += 1
            node.right, i = self.dfs(s, i)
            i += 1

        return node, i
```

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".

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
    def compress(self, chars: List[str]) -> int:
        d=[]
        c=1
        for i in range(1,len(chars)):
            if chars[i]==chars[i-1]:
                c+=1
            else:
                d.append([chars[i-1],c])
                c=1
        d.append([chars[-1],c])
        i=0
        for k,v in d:
            chars[i]=k
            i+=1
            if v>1:
                for item in str(v):
                    chars[i]=str(item)
                    i+=1
        return i
```

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".

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        hm, res, pL, sL = defaultdict(int), [], len(p), len(s)
        if pL > sL: return []

        # build hashmap
        for ch in p: hm[ch] += 1

        # initial full pass over the window
        for i in range(pL-1):
            if s[i] in hm: hm[s[i]] -= 1

        # slide the window with stride 1
        for i in range(-1, sL-pL+1):
            if i > -1 and s[i] in hm:
                hm[s[i]] += 1
            if i+pL < sL and s[i+pL] in hm:
                hm[s[i+pL]] -= 1

            # check whether we encountered an anagram
            if all(v == 0 for v in hm.values()):
                res.append(i+1)

        return res
```

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"

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
    def decodeString(self, s: str) -> str:
        st = []
        num = 0
        res = ''

        for ch in s:
            if ch.isnumeric():
                num = num * 10 + int(ch)
            elif ch == '[':
                st.append(res)
                st.append(num)
                res = ''
                num = 0
            elif ch == ']':
                cnt = st.pop()
                prev = st.pop()
                res = prev + cnt * res
            else:
                res += ch
        return res
```

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.

SOLUTIONS:

TC:O(n), SC:O(1)

CODE:

```
class Solution:
    def buddyStrings(self, A: str, B: str) -> bool:
        # goal: make one swap to A sp that A == B

        # length
        if len(A) != len(B):
            return False

        # A == B condition
        if A == B and len(set(A)) < len(A):
            return True

        # The length is same so now we check weather or not we can reach B
by making a single swap
        differences = []
        for x in range(len(B)):
            if A[x] != B[x]:
                differences.append([A[x], B[x]])

        if len(differences) == 2 and differences[0] == differences[-1][::-
1]:
            return True

        return False
```