Q1Given two strings s1 and s2, return *the lowest* ***ASCII*** *sum of deleted characters to make two strings equal*.

class Solution:

def minimumDeleteSum(self, s1: str, s2: str) -> int:

dp = [[0 for j in range(len(s2)+1)] for i in range(len(s1)+1)]

for i in range(len(s1)-1,-1,-1):

for j in range(len(s2)-1,-1,-1):

if s1[i] == s2[j]:

dp[i][j] = ord(s1[i]) + dp[i+1][j+1]

else:

dp[i][j] = max(dp[i+1][j],dp[i][j+1])

total = 0

for c in s1:

total += ord(c)

for c in s2:

total += ord(c)

return total - dp[0][0]\*2

q2. 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 "".

class Solution:

def checkValidString(self, s: str) -> bool:

left\_paren\_stack, asterisk\_stack = [], []

for i in range(len(s)):

if s[i] == '(':

left\_paren\_stack.append(i)

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

asterisk\_stack.append(i)

else:

if left\_paren\_stack:

left\_paren\_stack.pop()

elif asterisk\_stack:

asterisk\_stack.pop()

else:

return False

while left\_paren\_stack and asterisk\_stack:

if left\_paren\_stack[-1] > asterisk\_stack[-1]:

return False

left\_paren\_stack.pop()

asterisk\_stack.pop()

return not left\_paren\_stack

Q3. Given two strings word1 and word2, return *the minimum number of* ***steps*** *required to make* word1 *and* word2 *the same*.

class Solution:

def minDistance(self, word1: str, word2: str) -> int:

m=len(word1)

n=len(word2)

dp=[]

for i in range (m+1):

dp.append([0]\*(n+1))

for i in range (m+1):

dp[i][0]=i

for i in range (n+1):

dp[0][i]=i

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][j-1],dp[i-1][j])+1

return dp[-1][-1]

Q4. 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.

class newNode:

def \_\_init\_\_(self, data):

self.data = data

self.left = self.right = None

def preOrder(node):

if (node == None):

return

print(node.data, end=' ')

preOrder(node.left)

preOrder(node.right)

def findIndex(Str, si, ei):

if (si > ei):

return -1

s = []

for i in range(si, ei + 1):

if (Str[i] == '('):

s.append(Str[i])

elif (Str[i] == ')'):

if (s[-1] == '('):

s.pop(-1)

if len(s) == 0:

return i

return -1

def treeFromString(Str, si, ei):

if (si > ei):

return None

root = newNode(ord(Str[si]) - ord('0'))

index = -1

if (si + 1 <= ei and Str[si + 1] == '('):

index = findIndex(Str, si + 1, ei)

if (index != -1):

root.left = treeFromString(Str, si + 2,

index - 1)

root.right = treeFromString(Str, index + 2,

ei - 1)

return root

<aside>

Q5. 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.

class Solution:

def compress(self, chars: List[str]) -> int:

i=0

count=1

while i<len(chars)-1:

if chars[i+1]==chars[i]:

chars.pop(i+1)

count+=1

elif count>1:

cc=[\*str(count)]

for j in range(len(cc)):

chars.insert(i+j+1,cc[j])

count=1

i+=len(cc)+1

else:

i+=1

if count>1:

chars+=[\*str(count)]

return len(chars)

Q6. 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 findAnagrams(self, s: str, p: str) -> List[int]:

hm, res, pL, sL = defaultdict(int), [], len(p), len(s)

if pL > sL: return []

for ch in p: hm[ch] += 1

for i in range(pL-1):

if s[i] in hm: hm[s[i]] -= 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

if all(v == 0 for v in hm.values()):

res.append(i+1)

return res

Q7. 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.

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

Q8. 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].

class Solution:

def buddyStrings(self, s: str, goal: str) -> bool:

if len(s) != len(goal):

return False

if s == goal:

return len(set(s)) < len(s)

diffs = []

for i in range(len(s)):

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

diffs.append((s[i], goal[i]))

return len(diffs) == 2 and diffs[0] == diffs[1][::-1]