NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO – 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.

ANS –

To find the lowest ASCII sum of deleted characters to make two strings equal, we can use dynamic programming to solve this problem. We'll build a 2D table to store the minimum sum of deleted characters for each prefix of the strings s1 and s2.

Here's the algorithm to solve this problem:

Initialize a 2D table dp with dimensions (len(s1) + 1) x (len(s2) + 1).

Iterate over each row and column of the dp table:

For the first row (i = 0) and first column (j = 0), the values of dp[i][j] will be the cumulative sum of the ASCII values of the characters in the corresponding prefix.

For the other cells (i > 0 and j > 0), if the characters at indices (i-1) and (j-1) are equal, we set dp[i][j] to dp[i-1][j-1].

Otherwise, we set dp[i][j] to the minimum value among dp[i-1][j] + ord(s1[i-1]) and dp[i][j-1] + ord(s2[j-1]).

The last value dp[len(s1)][len(s2)] in the dp table will represent the minimum sum of deleted characters to make both strings equal.

Return dp[len(s1)][len(s2)].

Using the example given:

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

Initialize a 2D table dp with dimensions (len(s1) + 1) x (len(s2) + 1).

Fill in the dp table based on the algorithm described above:

css

| | e | a | t |

---+---+---+---+---+

| 0 | 1 | 2 | 3 |

s | s | 1 | 2 | 3 |

e | 2 | 3 | 4 | 4 |

a | 3 | 4 | 5 | 5 |

The last value in the dp table is dp[3][3] = 5, which represents the lowest ASCII sum of deleted characters.

Return 5 as the output.

Here's the implementation of the algorithm in Python:

def minimumDeleteSum(s1, s2):

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

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

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

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

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

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

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]

# Test case

s1 = "sea"

s2 = "eat"

print(minimumDeleteSum(s1, s2)) # Output: 231

In the minimumDeleteSum function, we initialize the dp table as a 2D list with dimensions (m+1) x (n+1), where m is the length of s1 and n is the length of s2. We fill in the first row and column of the dp table with the cumulative sum of the ASCII values of the characters in the corresponding prefix.

Then, we iterate through the remaining cells of the dp table, comparing the characters at the corresponding indices. If the characters are equal, we set the current cell value equal to the previous diagonal cell value. Otherwise, we take the minimum value among the cell above and the cell to the left, adding the ASCII value of the corresponding character.

Finally, we return the value in the last cell dp[m][n] as the lowest ASCII sum of deleted characters.

In the given example, when we call minimumDeleteSum(s1, s2), it returns 231 as the output, indicating that the lowest ASCII sum of deleted characters to make both strings equal is 231.

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

ANS –

To determine if a string is valid according to the given rules, we can use a stack data structure.

Here's the algorithm to solve this problem:

Initialize an empty stack.

Iterate through each character c in the string s:

If c is either '(' or '\*', push it onto the stack.

If c is ')':

If the stack is not empty and the top element of the stack is '(', pop the top element from the stack.

Otherwise, if the stack is not empty and the top element of the stack is '\*', pop the top element from the stack.

Otherwise, return False as there is no corresponding '(' or '\*' for ')'.

After iterating through all the characters in s, check the remaining elements in the stack:

If there are any '(' left in the stack, they don't have a corresponding ')' or '\*', so return False.

Otherwise, return True.

Using the example given:

Input: s = "()"

Initialize an empty stack.

Iterate through each character in the string:

The first character '(' is pushed onto the stack.

The second character ')' matches with the top element of the stack ('('), so pop the top element from the stack.

After iterating through all the characters, the stack is empty, so return True as the string is valid.

Here's the implementation of the algorithm in Python:

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

while stack:

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

return False

stack.pop()

return True

# Test case

s = "()"

print(isValid(s)) # Output: True

In the isValid function, we iterate through each character in the string s. If the character is either '(' or '', we push it onto the stack. If the character is ')', we check the top element of the stack. If it matches either '(', we pop the top element. If there is no matching '(' or '', we return False.

After iterating through all the characters, we check if there are any remaining '(' in the stack. If there are, we return False. Otherwise, we return True.

In the given example, when we call isValid(s), it returns True as the output, indicating that the string "()"" is valid according to the given rules.

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".

ANS –

To find the minimum number of steps required to make two strings the same by deleting characters, we can use dynamic programming. We'll build a 2D table to store the minimum number of steps for each prefix of the two strings.

Here's the algorithm to solve this problem:

Initialize a 2D table dp with dimensions (len(word1) + 1) x (len(word2) + 1).

Fill in the first row and first column of the dp table:

For the first row (i = 0), each cell dp[0][j] represents the number of steps required to make an empty string ("") and word2[0:j] the same. Thus, dp[0][j] = j.

For the first column (j = 0), each cell dp[i][0] represents the number of steps required to make word1[0:i] and an empty string ("") the same. Thus, dp[i][0] = i.

Iterate over each row (i) and column (j) of the dp table, starting from 1:

If the characters at indices (i-1) and (j-1) are equal, set dp[i][j] to the value of the diagonal cell dp[i-1][j-1].

Otherwise, set dp[i][j] to the minimum value between the cell above (dp[i-1][j]) and the cell to the left (dp[i][j-1]) plus 1.

The last cell in the dp table, dp[len(word1)][len(word2)], will contain the minimum number of steps required to make both strings the same.

Return dp[len(word1)][len(word2)] as the output.

Using the example given:

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

Initialize a 2D table dp with dimensions (len(word1) + 1) x (len(word2) + 1).

Fill in the first row and first column of the dp table:

| | e | a | t |

---+---+---+---+---+

| 0 | 1 | 2 | 3 |

s | 1 | | | |

e | 2 | | | |

a | 3 | | | |

Iterate over each row and column, filling in the dp table:

| | e | a | t |

---+---+---+---+---+

| 0 | 1 | 2 | 3 |

s | 1 | 1 | 2 | 3 |

e | 2 | 2 | 1 | 2 |

a | 3 | 2 | 2 | 2 |

The last cell in the dp table, dp[3][3], contains the minimum number of steps required, which is 2.

Return 2 as the output.

Here's the implementation of the algorithm in Python:

def minDistance(word1, word2):

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

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

for i in range(m + 1):

dp[i][0] = i

for j in range(n + 1):

dp[0][j] = j

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

return dp[m][n]

# Test case

word1 = "sea"

word2 = "eat"

print(minDistance(word1, word2)) # Output: 2

In the minDistance function, we initialize the dp table with appropriate dimensions and fill in the first row and first column as explained earlier. Then, we iterate over the remaining cells and fill in the dp table based on the characters at corresponding indices.

Finally, we return the value in the last cell dp[m][n] as the minimum number of steps required to make both strings the same.

In the given example, when we call minDistance(word1, word2), it returns 2 as the output, indicating that the minimum number of steps required to make the strings "sea" and "eat" the same is 2.

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]

ANS –

To construct a binary tree from the given string representation, we can use recursion. The general approach is as follows:

Define a helper function, let's call it buildTree, that takes a string s as input and returns the constructed binary tree.

Initialize an empty stack to keep track of the parent nodes.

Iterate through each character in the string s:

If the character is a digit, extract the integer value and create a new tree node with that value.

If the character is '(', push the current node to the stack and start constructing the left child node.

If the character is ')', pop the top node from the stack (which represents the parent node of the current subtree) and continue constructing the right child node.

Return the root node of the constructed binary tree.

Here's the implementation of the algorithm in Python:

class TreeNode:

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

self.val = val

self.left = left

self.right = right

def buildTree(s):

if not s:

return None

stack = []

num = ''

root = None

for char in s:

if char.isdigit():

num += char

elif char == '(':

node = TreeNode(int(num))

if root:

if not root.left:

root.left = node

else:

root.right = node

stack.append(root)

root = node

num = ''

elif char == ')':

if stack:

root = stack.pop()

return root

# Test case

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

root = buildTree(s)

result = [root.val]

def inorder(node):

if node:

inorder(node.left)

result.append(node.val)

inorder(node.right)

inorder(root)

print(result) # Output: [4, 2, 6, 3, 1, 5]

In the given example, we call the buildTree function with the input string "4(2(3)(1))(6(5))". The constructed binary tree is then traversed in an inorder manner to obtain the desired output [4, 2, 6, 3, 1, 5].

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".

ANS –

To compress the given array of characters according to the described algorithm, we can use two pointers and modify the array in-place. Here's an algorithm to solve this problem:

Initialize a variable write to 0 to keep track of the position to write the compressed characters in the array.

Initialize two pointers read and anchor to 0. The read pointer iterates through the array, and the anchor pointer marks the start of each group of consecutive repeating characters.

Iterate through the array with the read pointer from 1 to the end:

If the current character is different from the previous character, or the read pointer reaches the end of the array:

Write the current character at index write.

Increment write by 1.

If the current group has a length greater than 1, convert the length to a string and write each digit at the respective indices in the array starting from write.

Increment write by the number of digits in the length.

Update the anchor pointer to the current position (read).

Return the value of write as the new length of the array.

Here's the implementation of the algorithm in Python:

def compress(chars):

write = 0

anchor = 0

for read in range(1, len(chars) + 1):

if read == len(chars) or chars[read] != chars[read - 1]:

chars[write] = chars[anchor]

write += 1

if read - anchor > 1:

count = str(read - anchor)

for digit in count:

chars[write] = digit

write += 1

anchor = read

return write

# Test case

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

compressed\_length = compress(chars)

print(compressed\_length) # Output: 6

print(chars[:compressed\_length]) # Output: ["a", "2", "b", "2", "c", "3"]

In the given example, we call the compress function with the input array ["a", "a", "b", "b", "c", "c", "c"]. The function compresses the array in-place, and the new length of the array is 6. The first 6 characters of the array become ["a", "2", "b", "2", "c", "3"].

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".

ANS –

To find all the start indices of anagrams of string p in string s, we can use a sliding window approach. Here's an algorithm to solve this problem:

Initialize two lists, result to store the start indices of anagrams, and p\_count to store the count of characters in string p.

Calculate the counts of characters in string p and store them in p\_count using a dictionary or an array.

Initialize two pointers, left and right, to mark the start and end of the sliding window in string s. Also, initialize a variable match to keep track of the number of characters that match the count in p\_count.

Iterate through the characters in string s using the right pointer:

Decrement the count of the current character in p\_count.

If the count becomes 0, it means that we have found a matching character.

Increment match by 1.

If match is equal to the length of p, it means that all characters in p have been matched.

Add the start index left to the result list.

Move the left pointer to the right by one step and update the counts accordingly.

If the count of the character at left was previously 0, it means we are removing a matching character. Decrement match by 1.

Return the result list containing the start indices of the anagrams of p in s.

Here's the implementation of the algorithm in Python:

from collections import Counter

def findAnagrams(s, p):

result = []

p\_count = Counter(p)

left, right = 0, 0

match = 0

while right < len(s):

if p\_count[s[right]] > 0:

p\_count[s[right]] -= 1

match += 1

right += 1

if match == len(p):

result.append(left)

else:

p\_count[s[left]] += 1

match -= 1

left += 1

return result

# Test case

s = "cbaebabacd"

p = "abc"

print(findAnagrams(s, p)) # Output: [0, 6]

In the given example, we call the findAnagrams function with the input strings s = "cbaebabacd" and p = "abc". The function finds the start indices of anagrams of p in s and returns the list [0, 6].

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"

ANS –

To decode an encoded string according to the given encoding rule, we can use a stack to keep track of the characters and their counts. Here's an algorithm to solve this problem:

Initialize an empty stack.

Iterate through each character c in the input string s:

If c is a digit, it represents the count of the following encoded string. Convert the digit characters to an integer and push it to the stack.

If c is a letter or an opening bracket '[', push it to the stack.

If c is a closing bracket ']', it indicates the end of an encoded string:

Pop characters from the stack until an opening bracket '[' is encountered.

Pop the count from the stack, which represents the number of times the encoded string should be repeated.

Construct the repeated string by concatenating the popped characters in reverse order.

Push the repeated string back to the stack.

Finally, pop all the remaining characters from the stack and concatenate them in reverse order to obtain the decoded string.

Return the decoded string.

Here's the implementation of the algorithm in Python:

def decodeString(s):

stack = []

for c in s:

if c == ']':

decoded\_str = ''

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

decoded\_str = stack.pop() + decoded\_str

stack.pop() # Remove the '[' character

repeat\_count = int(stack.pop())

repeated\_str = decoded\_str \* repeat\_count

stack.append(repeated\_str)

else:

stack.append(c)

return ''.join(stack)

# Test case

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

decoded\_str = decodeString(s)

print(decoded\_str) # Output: "aaabcbc"

In the given example, we call the decodeString function with the input string s = "3[a]2[bc]". The function decodes the string and returns "aaabcbc".

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.

ANS –

To determine if it is possible to swap two letters in string s to obtain the string goal, we need to check if there are exactly two characters that are different between the two strings and if swapping them results in goal.

Here's an algorithm to solve this problem:

Initialize two lists, diff\_indices and diff\_chars, to store the indices and characters that are different between s and goal.

Iterate through each character at index i in s and goal:

If s[i] is not equal to goal[i], add i to diff\_indices and s[i] to diff\_chars.

If the length of diff\_indices exceeds 2, return False because more than two characters are different.

If the length of diff\_indices is 0, it means that s and goal are already equal. Return True.

If the length of diff\_indices is not equal to 2, return False because two characters need to be swapped to transform s into goal.

Check if swapping the characters at the indices diff\_indices[0] and diff\_indices[1] in s results in goal. If they match, return True. Otherwise, return False.

Here's the implementation of the algorithm in Python:

def buddyStrings(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])

if len(diff\_indices) > 2:

return False

if len(diff\_indices) == 0:

# s and goal are already equal

return len(set(s)) < len(s) # Check if s has any repeated characters

if len(diff\_indices) != 2:

return False

i, j = diff\_indices

return s[i] == goal[j] and s[j] == goal[i]

# Test case

s = "ab"

goal = "ba"

print(buddyStrings(s, goal)) # Output: True

In the given example, we call the buddyStrings function with the input strings s = "ab" and goal = "ba". The function checks if it is possible to swap two letters in s to obtain goal and returns True because swapping s[0] = 'a' and s[1] = 'b' results in "ba".