NAME – AMEY HUSHAR PATKAR

ASSIGNMENT NO – 7

Question 1

Given two strings s and t, determine if they are isomorphic.

Two strings s and t are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

Input: s = "egg", t = "add"

Output: true

ANS –

To determine if two strings are isomorphic, we need to check if we can replace the characters in the first string (s) to obtain the second string (t), while preserving the order of characters and ensuring that no two characters map to the same character.

One approach to solving this problem is by using a hash map to keep track of the character mappings. We iterate through each character in the strings and compare their corresponding mappings.

Here's the algorithm to determine if two strings are isomorphic:

If the lengths of s and t are not equal, return false.

Create two empty hash maps: s\_to\_t and t\_to\_s. These maps will store the character mappings from s to t and from t to s, respectively.

Iterate over each character at index i from 0 to length - 1:

Let char\_s be the character at index i in s, and char\_t be the character at index i in t.

Check if char\_s is already mapped to a character other than char\_t in s\_to\_t map, or if char\_t is already mapped to a character other than char\_s in t\_to\_s map. If either condition is true, return false.

If char\_s is not already in s\_to\_t map, add the mapping char\_s to char\_t.

If char\_t is not already in t\_to\_s map, add the mapping char\_t to char\_s.

If the loop completes without returning false, return true.

Using the example given:

Input: s = "egg", t = "add"

The lengths of both strings are equal, so we proceed with the algorithm.

Iteration 1:

char\_s = 'e', char\_t = 'a'

Neither char\_s nor char\_t is present in the maps, so we add the mappings e->a and a->e to the maps.

Iteration 2:

char\_s = 'g', char\_t = 'd'

Neither char\_s nor char\_t is present in the maps, so we add the mappings g->d and d->g to the maps.

Iteration 3:

char\_s = 'g', char\_t = 'd'

char\_s is already present in the s\_to\_t map and mapped to 'a', and char\_t is already present in the t\_to\_s map and mapped to 'e'. Both conditions are satisfied.

Since we have successfully completed the loop without returning false, we conclude that the strings "egg" and "add" are isomorphic. The output is true.

Here's the fully implemented solution in Python:

def isomorphic\_strings(s, t):

if len(s) != len(t): # Check if the lengths of s and t are equal

return False

s\_to\_t = {} # Map characters from s to t

t\_to\_s = {} # Map characters from t to s

for i in range(len(s)):

char\_s = s[i]

char\_t = t[i]

if char\_s in s\_to\_t:

if s\_to\_t[char\_s] != char\_t:

return False

else:

s\_to\_t[char\_s] = char\_t

if char\_t in t\_to\_s:

if t\_to\_s[char\_t] != char\_s:

return False

else:

t\_to\_s[char\_t] = char\_s

return True

# Test case

s = "egg"

t = "add"

print(isomorphic\_strings(s, t)) # Output: True

The isomorphic\_strings function takes two strings s and t as input and returns True if they are isomorphic, and False otherwise.

In this function, we first check if the lengths of s and t are equal. If they are not equal, the function immediately returns False since the strings cannot be isomorphic.

We then create two empty dictionaries, s\_to\_t and t\_to\_s, which will store the character mappings from s to t and from t to s, respectively.

Next, we iterate through each character in s and t using the for loop. For each character, we check if it is already mapped to a different character in either s\_to\_t or t\_to\_s dictionaries. If it is, we return False since the character mappings would violate the isomorphic condition.

If the character is not already in the respective dictionary, we add the character mappings. For example, if char\_s is not in s\_to\_t, we add the mapping char\_s to char\_t, and if char\_t is not in t\_to\_s, we add the mapping char\_t to char\_s.

Finally, if we complete the loop without returning False, it means that the strings are isomorphic, so we return True.

In the given example, when we call isomorphic\_strings(s, t), it returns True since the strings "egg" and "add" can be transformed by replacing 'e' with 'a' and 'g' with 'd' to obtain the second string.

Question 2

Given a string num which represents an integer, return true if num is a strobogrammatic number.

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Example 1:

Input: num = "69"

Output:

True

ANS -

To determine if a given number is a strobogrammatic number, we need to check if it looks the same when rotated 180 degrees, which means the digits in the number should have valid rotations.

Here's the algorithm to check if a number is strobogrammatic:

Initialize two pointers, left and right, to the start and end of the string num, respectively.

While left is less than or equal to right:

Get the characters at positions left and right in num.

If the pair of characters is not a valid strobogrammatic digit pair (0 and 0, 1 and 1, 6 and 9, 8 and 8, or 9 and 6), return false.

Increment left and decrement right.

If the loop completes without returning false, return true.

Using the example given:

Input: num = "69"

We start with left = 0 and right = 1.

Iteration 1:

char\_left = '6', char\_right = '9'

The pair (6, 9) is a valid strobogrammatic digit pair.

Since we have successfully completed the loop without returning false, we conclude that the number "69" is a strobogrammatic number. The output is true.

Here's the implementation of the algorithm in Python:

def is\_strobogrammatic(num):

strobogrammatic\_pairs = {'0': '0', '1': '1', '6': '9', '8': '8', '9': '6'}

left, right = 0, len(num) - 1

while left <= right:

char\_left = num[left]

char\_right = num[right]

if char\_left not in strobogrammatic\_pairs or strobogrammatic\_pairs[char\_left] != char\_right:

return False

left += 1

right -= 1

return True

# Test case

num = "69"

print(is\_strobogrammatic(num)) # Output: True

In the is\_strobogrammatic function, we first define a dictionary strobogrammatic\_pairs that stores the valid strobogrammatic digit pairs.

We then initialize the pointers left and right to the start and end of the string num, respectively.

Inside the while loop, we get the characters at positions left and right in num and check if the pair is a valid strobogrammatic digit pair. If the pair is not valid, we return False.

If the pair is valid, we increment left and decrement right, and continue checking the next pair.

If the loop completes without returning false, it means all pairs are valid strobogrammatic digit pairs, so we return True.

In the given example, when we call is\_strobogrammatic(num), it returns True since the number "69" is a strobogrammatic number when rotated 180 degrees.

Question 3

Given two non-negative integers, num1 and num2 represented as string, return the sum of num1 and num2 as a string.

You must solve the problem without using any built-in library for handling large integers (such as BigInteger). You must also not convert the inputs to integers directly.

Example 1:

Input: num1 = "11", num2 = "123"

Output:

"134"

ANS –

To add two non-negative integers represented as strings, we can perform the addition manually digit by digit. We start from the least significant digit and work our way up, keeping track of the carry if it occurs.

Here's the algorithm to add two non-negative integer strings:

Initialize an empty string result to store the final sum.

Initialize two pointers, i and j, to the end of the input strings num1 and num2, respectively.

Initialize a carry variable carry to 0.

While i and j are greater than or equal to 0:

Convert the characters at positions i and j in num1 and num2 to integers: digit1 and digit2, respectively.

Calculate the sum: digit\_sum = digit1 + digit2 + carry.

Append the least significant digit of digit\_sum (i.e., digit\_sum % 10) to the front of result.

Update the carry: carry = digit\_sum // 10.

Decrement i and j by 1.

If there is a remaining carry, append it to the front of result.

Return the final result string.

Using the example given:

Input: num1 = "11", num2 = "123"

We start with i = 1, j = 2, and an empty result string.

Iteration 1:

digit1 = 1, digit2 = 3

digit\_sum = 1 + 3 + 0 = 4

Append the least significant digit, 4 % 10 = 4, to the front of result: "4"

carry = 4 // 10 = 0

Decrement i and j by 1: i = 0, j = 1

Iteration 2:

digit1 = 1, digit2 = 2

digit\_sum = 1 + 2 + 0 = 3

Append the least significant digit, 3 % 10 = 3, to the front of result: "34"

carry = 3 // 10 = 0

Decrement i and j by 1: i = -1, j = 0

Both pointers i and j are less than 0, so we stop the loop.

Since there is no remaining carry, the final result is "34". The output is "34".

Here's the implementation of the algorithm in Python:

def addStrings(num1, num2):

i, j = len(num1) - 1, len(num2) - 1

carry = 0

result = ""

while i >= 0 or j >= 0:

digit1 = int(num1[i]) if i >= 0 else 0

digit2 = int(num2[j]) if j >= 0 else 0

digit\_sum = digit1 + digit2 + carry

result = str(digit\_sum % 10) + result

carry = digit\_sum // 10

i -= 1

j -= 1

if carry > 0:

result = str(carry) + result

return result

# Test case

num1 = "11"

num2 = "123"

print(addStrings(num1, num2)) # Output: "134"

In the addStrings function, we initialize the pointers i and j to the end of num1 and num2, respectively. We also initialize the carry variable carry to 0 and the result string result to an empty string.

Inside the while loop, we convert the characters at positions i and j in num1 and num2 to integers. We calculate the sum of the digits, append the least significant digit of the sum to the front of the result string, and update the carry accordingly. Then, we decrement i and j by 1.

After the loop, if there is a remaining carry, we append it to the front of the result string.

Finally, we return the result string as the sum of num1 and num2.

In the given example, when we call addStrings(num1, num2), it returns "134" as the sum of the input strings "11" and "123".

Question 4

Given a string s, reverse the order of characters in each word within a sentence while still preserving whitespace and initial word order.

Example 1:

Input: s = "Let's take LeetCode contest"

Output: "s'teL ekat edoCteeL tsetnoc"

ANS –

To reverse the order of characters in each word within a sentence while preserving whitespace and the initial word order, we can follow these steps:

Split the input string s into individual words using whitespace as the delimiter. We can use the split() function in Python for this.

Iterate through each word and reverse its order of characters.

Join the reversed words back together using whitespace as the separator. We can use the join() function in Python for this.

Here's the implementation of the algorithm in Python:

def reverseWords(s):

words = s.split() # Split the input string into words

reversed\_words = [word[::-1] for word in words] # Reverse each word

return ' '.join(reversed\_words) # Join the reversed words back together with whitespace

# Test case

s = "Let's take LeetCode contest"

print(reverseWords(s)) # Output: "s'teL ekat edoCteeL tsetnoc"

In the reverseWords function, we first split the input string s into individual words using the split() function. This creates a list of words.

Next, we iterate through each word in the list using a list comprehension. For each word, we use slicing with a step of -1 (word[::-1]) to reverse the order of its characters.

Finally, we join the reversed words back together using the join() function, specifying whitespace as the separator.

In the given example, when we call reverseWords(s), it returns "s'teL ekat edoCteeL tsetnoc" as the output, which is the input string with each word's characters reversed while preserving whitespace and the initial word order.

Question 5

Given a string s and an integer k, reverse the first k characters for every 2k characters counting from the start of the string.

If there are fewer than k characters left, reverse all of them. If there are less than 2k but greater than or equal to k characters, then reverse the first k characters and leave the other as original.

Example 1:

Input: s = "abcdefg", k = 2

Output:

"bacdfeg"

ANS –

To reverse the first k characters for every 2k characters counting from the start of the string, we can iterate through the string with a step size of 2k. For each group of 2k characters, we reverse the first k characters if there are k or more characters left. Otherwise, we reverse all the remaining characters.

Here's the algorithm to reverse the characters based on the given conditions:

Convert the string s to a list of characters so that we can modify it. (Strings in Python are immutable.)

Initialize a variable i to 0 to keep track of the current position in the string.

While i is less than the length of the string s:

Reverse the characters from index i to i+k if there are k or more characters left. Otherwise, reverse all the remaining characters.

Update i by adding 2k.

Convert the modified list back to a string and return it.

Using the example given:

Input: s = "abcdefg", k = 2

We start with i = 0.

Iteration 1:

Reverse the characters from index 0 to 2: "bac"

Update i to 4

Iteration 2:

There are fewer than 4 characters left, so reverse all the remaining characters: "bacdfeg"

Update i to 8

Since i is now greater than or equal to the length of the string, we stop the loop.

The final string is "bacdfeg". The output is "bacdfeg".

Here's the implementation of the algorithm in Python:

def reverseStr(s, k):

s = list(s) # Convert the string to a list of characters

i = 0 # Start at index 0

while i < len(s):

if i + k <= len(s):

s[i:i+k] = reversed(s[i:i+k]) # Reverse the first k characters

else:

s[i:] = reversed(s[i:]) # Reverse all the remaining characters

i += 2 \* k # Update i by adding 2k

return ''.join(s) # Convert the list back to a string

# Test case

s = "abcdefg"

k = 2

print(reverseStr(s, k)) # Output: "bacdfeg"

In the reverseStr function, we first convert the input string s to a list of characters using the list() function. This allows us to modify the characters in place.

We then initialize i to 0 and start the while loop. For each iteration, we check if there are k or more characters remaining from the current position i. If there are, we use slicing to reverse the characters from index i to i+k using the reversed() function. Otherwise, we reverse all the remaining characters from index i to the end of the string.

After each iteration, we update i by adding 2k to move to the next group of 2k characters.

Finally, we join the modified list of characters back into a string using the join() function and return the result.

In the given example, when we call reverseStr(s, k), it returns "bacdfeg" as the output, which is the input string with the first k characters reversed for every 2k characters.

Question 6

Given two strings s and goal, return true if and only if s can become goal after some number of shifts on s.

A shift on s consists of moving the leftmost character of s to the rightmost position.

- For example, if s = "abcde", then it will be "bcdea" after one shift.

Example 1:

Input: s = "abcde", goal = "cdeab"

Output:

True

ANS –

To determine if a string s can become goal after some number of shifts, we can concatenate s with itself to create a new string s\_concat. Then, we can check if goal is a substring of s\_concat.

Here's the algorithm to solve this problem:

Check if the lengths of s and goal are equal. If they are not, return False.

Concatenate s with itself to create s\_concat.

Check if goal is a substring of s\_concat. If it is, return True. Otherwise, return False.

Using the example given:

Input: s = "abcde", goal = "cdeab"

The lengths of s and goal are both 5, so we proceed.

Concatenating s with itself gives us s\_concat = "abcdeabcde".

"cdeab" is a substring of s\_concat, so we return True.

Here's the implementation of the algorithm in Python:

def rotateString(s, goal):

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

return False

s\_concat = s + s

if goal in s\_concat:

return True

return False

# Test case

s = "abcde"

goal = "cdeab"

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

In the rotateString function, we first check if the lengths of s and goal are equal. If they are not, we return False because s cannot become goal through shifting.

Next, we concatenate s with itself using the + operator to create s\_concat.

Then, we use the in operator to check if goal is a substring of s\_concat. If it is, we return True. Otherwise, we return False.

In the given example, when we call rotateString(s, goal), it returns True as the output, indicating that s can become goal after some number of shifts.

Question 7

Given two strings s and t, return true if they are equal when both are typed into empty text editors. '#' means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

Example 1:

Input: s = "ab#c", t = "ad#c"

Output: true

Explanation:

Both s and t become "ac".

ANS –

To determine if two strings s and t are equal when typed into empty text editors, accounting for backspace characters ('#'), we can simulate the typing process using a stack.

Here's the algorithm to solve this problem:

Define a helper function processString that takes a string as input and returns the final string after processing all the characters.

Initialize two variables s\_processed and t\_processed by calling the processString function on strings s and t, respectively.

Check if s\_processed is equal to t\_processed. If they are equal, return True. Otherwise, return False.

Using the example given:

Input: s = "ab#c", t = "ad#c"

After processing string s, we get s\_processed = "ac".

After processing string t, we get t\_processed = "ac".

Since s\_processed is equal to t\_processed, we return True.

Here's the implementation of the algorithm in Python:

def backspaceCompare(s, t):

def processString(string):

stack = []

for char in string:

if char == '#':

if stack:

stack.pop()

else:

stack.append(char)

return ''.join(stack)

s\_processed = processString(s)

t\_processed = processString(t)

return s\_processed == t\_processed

# Test case

s = "ab#c"

t = "ad#c"

print(backspaceCompare(s, t)) # Output: True

In the backspaceCompare function, we define a helper function processString that takes a string as input. Inside this function, we initialize an empty stack. We iterate through each character in the string. If the character is '#' and the stack is not empty, we remove the last character from the stack (backspace operation). Otherwise, if the character is not '#', we push it onto the stack. Finally, we convert the stack into a string using the join() function.

In the main function, we call processString on both s and t to obtain the processed versions s\_processed and t\_processed. We then compare the two processed strings using the == operator. If they are equal, we return True; otherwise, we return False.

In the given example, when we call backspaceCompare(s, t), it returns True as the output, indicating that s and t are equal when typed into empty text editors.

Question 8

You are given an array coordinates, coordinates[i] = [x, y], where [x, y] represents the coordinate of a point. Check if these points make a straight line in the XY plane.

Example 1:

Input: coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]

Output: true

ANS –

To check if a given set of coordinates forms a straight line in the XY plane, we can calculate the slope between every pair of consecutive points. If all the slopes are the same, then the points lie on a straight line.

Here's the algorithm to solve this problem:

Calculate the slope between the first two points (x1, y1) and (x2, y2) using the formula: slope = (y2 - y1) / (x2 - x1).

Iterate through the remaining points starting from the third point.

Calculate the slope between the current point (xi, yi) and the previous point (xi-1, yi-1).

If the calculated slope is not equal to the initial slope calculated in step 1, return False.

If all the slopes are the same, return True.

Using the example given:

Input: coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]

Calculate the slope between the first two points (1,2) and (2,3):

slope = (3 - 2) / (2 - 1) = 1 / 1 = 1.

Iterate through the remaining points starting from the third point (3,4):

Calculate the slope between (3,4) and (2,3): slope = (4 - 3) / (3 - 2) = 1 / 1 = 1.

Calculate the slope between (4,5) and (3,4): slope = (5 - 4) / (4 - 3) = 1 / 1 = 1.

Calculate the slope between (5,6) and (4,5): slope = (6 - 5) / (5 - 4) = 1 / 1 = 1.

Calculate the slope between (6,7) and (5,6): slope = (7 - 6) / (6 - 5) = 1 / 1 = 1.

All the slopes are equal to 1, so we return True.

Here's the implementation of the algorithm in Python:

def checkStraightLine(coordinates):

x1, y1 = coordinates[0]

x2, y2 = coordinates[1]

initial\_slope = (y2 - y1) / (x2 - x1)

for i in range(2, len(coordinates)):

xi, yi = coordinates[i]

xi\_1, yi\_1 = coordinates[i - 1]

slope = (yi - yi\_1) / (xi - xi\_1)

if slope != initial\_slope:

return False

return True

# Test case

coordinates = [[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]]

print(checkStraightLine(coordinates)) # Output: True

In the checkStraightLine function, we first calculate the initial slope between the first two points (x1, y1) and (x2, y2). Then, we iterate through the remaining points starting from the third point. For each point, we calculate the slope between the current point (xi, yi) and the previous point (xi-1, yi-1). If any calculated slope is not equal to the initial slope, we return False. If all the slopes are the same, we return True.

In the given example, when we call checkStraightLine(coordinates), it returns `True