CAPSTONEPROJECT

**STRING TRANSFORMATION**

**CSA0695-** DESIGN ANALYSIS AND ALGORITHMS

FOR OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING



Supervisor

Dr. R. Dhanalakshmi

Done by

S.Sivanjali (192210508)

**STRING TRANSFORMATION**

**PROBLEM STATEMENT:** Given two strings s and t of equal length n. Can perform the following operation on the string s: Remove a suffix of s of length l where 0 < l < n and append it at the start of s. For example, let s = 'abcd' then in one operation you can remove the suffix 'cd' and append it in front of s making s = 'cdab'. You are also given an integer k. Return the number of ways in which s can be transformed into t in exactly k operations.Since the answer can be large, return it modulo 109 + 7.

Example 1:

Input: s = "abcd", t = "cdab", k = 2

**ABSTRACT:**

This paper explores the problem of finding the two strings, s and t, of equal length n, and an integer k. The task is to determine how many ways string s can be transformed into string t in exactly k operations, where each operation consists of removing a suffix from s and appending it to the start (cyclic shift). The approach involves checking how many cyclic permutations of s match t and whether these matches align with the exact number of operations specified by k.

**INTRODUCTION:**

In computational string problems, transforming one string into another through a series of operations is a common challenge. This problem focuses on transforming a given string s into another string t by performing a specific operation multiple times. The operation consists of removing a suffix from s and appending it to the start, effectively creating a cyclic shift. The goal is to determine how many ways s can be transformed into t in exactly k operations, where k is a given integer. Since the number of transformations can be large, the result is to be returned modulo \(10^9 + 7\). This problem explores the nature of cyclic permutations and modular arithmetic, making it an interesting case of pattern matching and mathematical manipulation.

This problem presents challenges related to efficiently managing cyclic shifts, leveraging modular arithmetic to handle rotations, and ensuring the solution is optimized for potentially large inputs. By understanding the nature of cyclic permutations and utilizing string concatenation techniques, we can derive a solution that identifies valid transformations and returns the result modulo 109 +7 to manage large numbers.

This paper aims to address the The problem of string transformation is a common computational task involving the manipulation of strings through a series of operations. In this specific case, you are given two strings, s and t, of equal length n, and an integer k. The goal is to determine how many different ways string s can be transformed into string t using exactly k cyclic shift operations. A cyclic shift operation involves removing a suffix from s and appending it to the front of the string. Since the number of possible transformations can be large, the result needs to be returned modulo (10^9 + 7). The challenge lies in balancing computational efficiency with handling the modular arithmetic, making it an interesting problem in string manipulation and modular computation.

**CODING:**

**C-programming**

#include <stdio.h>

#include <string.h>

#define MOD 1000000007

// Function to count transformations from s to t in exactly k operations

int count\_transformations(char s[], char t[], int k) {

int n = strlen(s);

// Create a doubled string to check all cyclic permutations

char doubled\_s[2 \* n + 1];

strcpy(doubled\_s, s);

strcat(doubled\_s, s); // doubled\_s = s + s

// Count the number of valid cyclic shifts

int count = 0;

for (int i = 0; i < n; i++) {

// Check if the substring of doubled\_s starting at i matches t

if (strncmp(&doubled\_s[i], t, n) == 0) {

// Check if this shift can be done in exactly k moves

if (k % n == i) {

count++;

}

}

}

// Return the count modulo 10^9 + 7

return count % MOD;

}

int main() {

// Example input

char s[] = "abcd";

char t[] = "cdab";

int k = 2;

// Calculate and print the number of transformations

int result = count\_transformations(s, t, k);

printf("%d\n", result); // Output: 1

return 0;

}

**OUTPUT:**



**COMPLEXITY ANALYSIS:**

**Time Complexity**:

Cyclic Permutations Check:

We loop over all starting positions i from 0 to n - 1, so the loop runs n times.

For each starting position i, we compare the substring doubled\_s[i:i+n] with t using the strncmp function. The comparison of two strings of length n takes \( O(n) \) in the worst case (when all characters match or need to be compared).

**Space Complexity**:

1. Doubled String:

The space required for doubled\_s is \( 2n + 1 \) to store the concatenated string s + s and the null terminator. This takes \( O(n) \) space.

2. Other Variables:

We use some constant space for variables like the integer count and loop indices, which are \( O(1) \).

Overall Space Complexity:

The space complexity is dominated by the doubled\_s array, which takes \( O(n) \).

**BEST CASE:**

In the context of the given problem, the best case scenario for the time complexity occurs when the number of valid transformations is minimized, i.e., when there are fewer or no valid cyclic permutations of s that match t. This would reduce the number of operations needed to determine the result.The time complexity for concatenating s with itself to form doubled\_s remains \( O(n) \), regardless of the input. if t matches only one specific cyclic permutation of s, or if no cyclic permutations of s match t, the algorithm would find a match early or conclude there are no matches. Specifically:

**Worst Case:**

n the context of the problem, the worst-case scenario for the time complexity arises when the algorithm has to perform the maximum number of operations to determine the number of valid cyclic transformations. In the worst case, where every possible cyclic permutation needs to be checked to see if it matches t, the algorithm performs a string comparison for each possible starting index in doubled\_s. For each of the n starting positions, it compares a substring of length n with t. Each comparison takes \( O(n) \), leading to a total time complexity of \( O(n^2) \) for checking all cyclic permutations.

**Average Case:**

To analyze the average case time complexity of the problem, we consider a scenario where we have an arbitrary distribution of strings and operations, and t might or might not match some of the cyclic permutations of s. On average, we expect that t will match some of the cyclic permutations of s or none at all. The exact number of matches will depend on the specific strings s and t. Suppose t matches m cyclic permutations of s, where m is on average a fraction of n. The average number of matches will be proportional to n, and the average time complexity for checking all starting positions remains \( O(n^2) \).

**Future Scope:**

The problem of transforming one string into another through cyclic shifts has various applications and potential areas for future exploration and improvement. Here are some areas where this problem could be expanded or refined. Extend the problem to consider not just cyclic shifts but a combination of different operations (e.g., cyclic shifts, reversals, rotations) and determine how these operations can be combined to achieve the transformation. These future directions not only enhance the understanding and applicability of cyclic shifts and string transformations but also contribute to more efficient and practical solutions in various domains.

**CONCLUSION:**

In conclusion, the problem of transforming a string s into another string t through exactly k cyclic shifts is a classic example of string manipulation involving cyclic permutations. The solution involves efficiently determining how many ways s can be transformed into t by performing cyclic shifts, with a focus on achieving the transformation in exactly k operations. Overall, understanding the problem's time and space complexities provides insight into its computational demands, while exploring future advancements can lead to more efficient and broadly applicable solutions.