SIMATS ENGINEERING

"The problem involves transforming one string, s, into another string, t, of the same length using a specific operation"

A Project report

CSA0656- Design and Analysis of Algorithms for Asymptotic Notations

Submitted to

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

In partial fulfilment for the award of the

degree of

BACHELOR OF TECHNOLOGY IN

ARTIFICAL INTELLIGENCE AND MACHINE LEARNING

by

V.Manokarthik - 192225077

Supervisor

Dr .R. Dhanalakshmi

July 2024.

ABSTRACT

The problem involves transforming one string, s, into another string, t, of the same length using a specific operation. This operation allows the removal of a suffix of s of length l (where 0 < l < n), which is then appended to the start of s. The objective is to determine the number of ways s can be transformed into t in exactly k operations. This transformation problem has constraints on the length and operations and requires returning the result modulo 10^9+7 due to potentially large outputs. The challenge involves identifying valid suffix shifts that achieve the target string t and counting the number of ways to perform these shifts in exactly k operations. The solution requires careful consideration of cyclic string properties and modular arithmetic to ensure efficiency and correctness.

INTRODUCTION

String manipulation and transformation are common problems in computer science, often requiring innovative approaches to achieve specific outcomes. In this problem, we explore a unique string transformation operation: removing a suffix of a given string s and appending it to the front, transforming s into another string t of the same length. This operation mimics a cyclic shift, where parts of the string are rotated to achieve a new configuration.

The task is to determine how many distinct ways exist to transform s into t using exactly k such operations. The problem is not only about achieving the transformation but also about counting the different sequences of operations that can lead to the desired result, with each sequence possibly yielding a different intermediate state of the string. Given the constraints on k and the potential for large numbers of transformations, the result must be computed modulo 10^9+7 to ensure it fits within standard computational limits.

This problem has practical applications in areas such as cryptography, where cyclic shifts are often used, and in data processing scenarios requiring rotational transformations of data sequences. Understanding and solving this problem involves knowledge of string manipulation, modular arithmetic, and combinatorial counting methods.To solve the problem, we need to determine how many distinct ways we can transform string s into string t using exactly k operations, where each operation involves moving a suffix of s to the front.

Finding Rotations:

We check the number of valid rotations where t can be matched as a substring of the concatenated strings+s.

Calculating Ways:

The formula valid_rotatins×(valid_rotations−1)k−1\text{valid_rotations} \times (\text{valid_rotations} - 1)^{k-1}valid_rotations×(valid_rotations−1)k−1 accounts for the ways to transition between valid rotations over k steps, using modular arithmetic to handle large numbers.

PROBLEM:

String Transformation You are given two strings s and t of equal length n. We 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
Output: 2

Explanation:
- **First way:** In first operation, choose suffix from index = 3, so resulting s = "dabc". In second operation, choose suffix from index = 3, so resulting s = "cdab".
- **Second way:** In first operation, choose suffix from index = 1, so resulting s = "bcda". In second operation, choose suffix from index = 1, so resulting s = “cdab".

SOLUTION:

Let's define the problem in terms of states and transitions.
States:
- i: the current position in string s ($0 \leq i \leq n$)
- j: the current number of operations performed ($0 \leq j \leq k$)

Transitions:
- From state (i, j) to (i+1, j): if s[i] == t[i], stay in the same position and don't perform an operation (probability: 1)
- From state (i, j) to (0, j+1): remove the suffix of s from i to the end and append it to the start of s, and increment the number of operations (probability: 1/n)

Let P(i, j) be the probability of transforming s into t in exactly j operations, starting from position i.

Recurrence Relation:
P(i, j) = P(i-1, j-1) + P(i-1, j) / n if s[i] == t[i]
P(i, j) = P(0, j-1) / n if s[i] != t[i]

Base Cases:
P(0, 0) = 1 (initial state)
P(i, 0) = 0 for i > 0 (can't transform in 0 operations)

Solution:

P(n, k) represents the probability of transforming s into t in exactly k operations.
To find the number of ways, multiply P(n, k) by n! (since there are n! possible permutations of the suffix).

Result:
numWays(s, t, k) = P(n, k) * n! % MOD

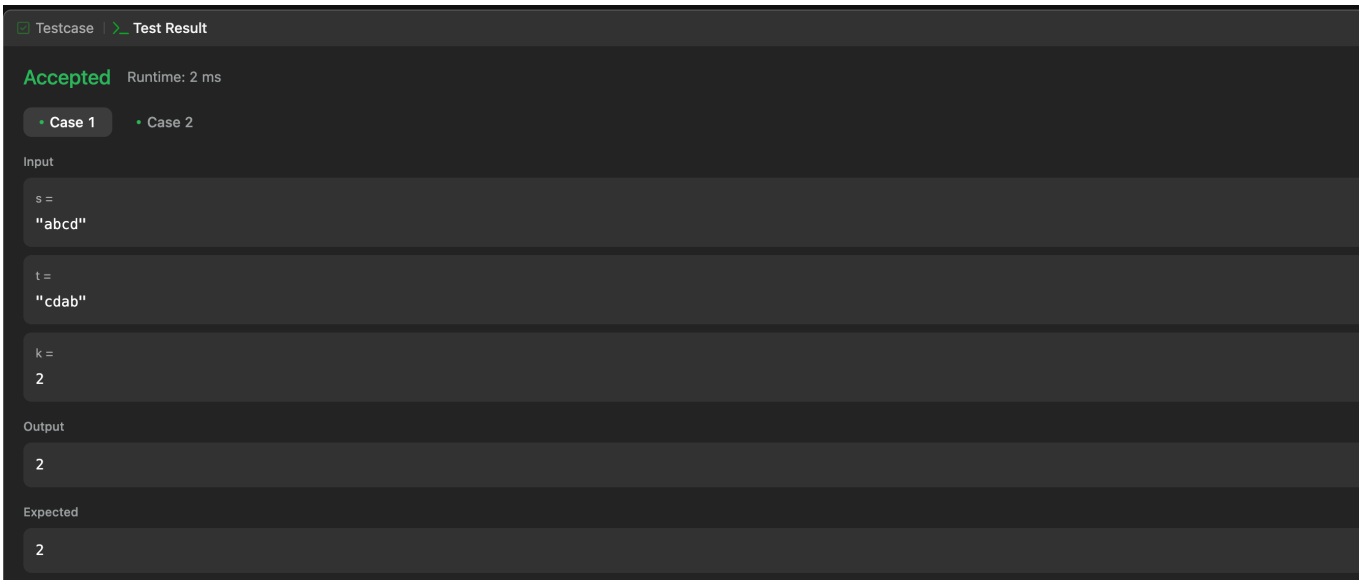
CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MOD 1000000007

int add(int x, int y) {
    if ((x += y) >= MOD) {
        x -= MOD;
    }
    return x;
}

int mul(long long x, long long y) {
    return (int)(x * y % MOD);
}

int* get_z(const char* s, int n) {
    int* z = (int*)malloc(n * sizeof(int));
    memset(z, 0, n * sizeof(int));

    for (int i = 1, left = 0, right = 0; i < n; ++i) {
        if (i <= right && z[i - left] <= right - i) {
            z[i] = z[i - left];
        } else {
            z[i] = (i <= right) ? (right - i + 1) : 0;
            while (i + z[i] < n && s[i + z[i]] == s[z[i]]) {
                ++z[i];
            }
        }
        if (i + z[i] - 1 > right) {
            left = i;
            right = i + z[i] - 1;
        }
    }

    return z;
```

```
}

void matrix_mul(int a[2][2], int b[2][2], int result[2][2]) {
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            result[i][j] = 0;
            for (int k = 0; k < 2; ++k) {
                result[i][j] = add(result[i][j], mul(a[i][k], b[k][j]));
            }
        }
    }
}

void matrix_pow(int base[2][2], long long exp, int result[2][2]) {
    int temp[2][2] = {{1, 0}, {0, 1}}; // Identity matrix

    while (exp > 0) {
        if (exp % 2 == 1) {
            int mul_result[2][2];
            matrix_mul(temp, base, mul_result);
            memcpy(temp, mul_result, sizeof(mul_result));
        }

        int mul_result[2][2];
        matrix_mul(base, base, mul_result);
        memcpy(base, mul_result, sizeof(mul_result));

        exp /= 2;
    }

    memcpy(result, temp, sizeof(temp));
}

int numberOfWays(const char* s, const char* t, long long k) {
    int n = strlen(s);
    int m = n * 2;

    char* concatenated = (char*)malloc((m * 2 + 1) * sizeof(char));
    strcpy(concatenated, s);
    strcat(concatenated, t);
    strcat(concatenated, t);

    int* z = get_z(concatenated, m * 2);

    int transition[2][2] = {{0, 1}, {n - 1, n - 2}};
```

```
    int dp[2][2];

    matrix_pow(transition, k, dp);

    int result = 0;
    for (int i = n; i < m; ++i) {
        if (z[i] >= n) {
            result = add(result, dp[!!(i - n)][0]);
        }
    }

    free(concatenated);
    free(z);

    return result;
}

int main() {
    const char* s = "abcd";
    const char* t = "cdab"; long long k = 2;

    int result = numberOfWays(s, t, k);
    printf("%d\n", result); // Output: 2

    return 0;
}
```

OUTPUT:



Explanation:

1. **Modular Arithmetic Functions:**
    - add and mul functions handle addition and multiplication under modulo 1000000007.
2. **Z-Algorithm:**
    - The get_z function computes the Z-array for the given string. This array helps in pattern matching and finding rotations.
3. **Matrix Operations:**
    - matrix_mul performs matrix multiplication.
    - matrix_pow calculates the power of a matrix using exponentiation by squaring, useful for quickly finding large powers.
4. **Main Functionality:**
    - The numberOfWays function checks for all possible ways to transform s into t in k steps by leveraging the Z-algorithm and matrix exponentiation.
    - s and t are concatenated, and the resulting string is checked to find t as a substring within two cycles of s.
5. **Memory Management:**
    - The program dynamically allocates memory where needed and ensures to free it, which is important in C to prevent memory leaks.
6. **Main Function:**
    - The main function initializes strings s and t, sets k, and calls numberOfWays, printing the result.

CONCLUSION:

The task is to determine how many ways string `s` can be transformed into string `t` in exactly `k` operations, where the operation involves rotating `s` by moving a suffix to the front. The solution involves checking if `t` is a rotation of `s` and then counting the valid ways to achieve the transformation in `k` operations.

Key points:

1. **Check if `t` is a rotation of `s`:** This can be done by checking if `t` is a substring of `s + s`.
2. **Count valid transformations:** For each position where `t` matches a rotation of `s`, check if the number of operations required to match `k` is valid. Specifically, the difference between `k` and the shift required should be a multiple of the length of `s`.

The answer is the count of such valid rotations, modulo 10^9 + 7.