

# **CAPSTONE PROJECT REPORT**

**Reg NO:192210719**

**Name: U. Karthik**

**Course Code: CSA0656**

**Course Name: Design And Analysis for Algorithm for Asymptotic Notations**

**SLOT: A**

## ABSTRACT

The problem of cracking a safe with a specific password checking mechanism involves finding a sequence of digits that unlocks the safe. The safe verifies the password by checking the most recent 'n' digits entered, where 'n' is a given parameter. The password itself is a sequence of 'n' digits in a specified range.

The goal is to design an algorithm that determines the shortest sequence of digits that guarantees the safe's unlocking. The algorithm needs to account for the safe's unique validation process and identify a sequence that triggers a successful match with the hidden password. The problem presents a challenge in finding an optimal strategy that minimizes the number of digits required to unlock the safe.

## INTRODUCTION

In this problem, you are tasked with unlocking a safe that is protected by a password. The password consists of a sequence of  $nnn$  digits, where each digit can be in the range  $[0, k-1][0, k-1][0, k-1]$ . The safe has a unique method for checking the password: it continuously checks the most recent  $nnn$  digits that were entered.

To illustrate, consider the example where the correct password is "345" and the sequence entered is "012345":

- After typing 0, the most recent 3 digits are "0", which is incorrect.
- After typing 1, the most recent 3 digits are "01", which is incorrect.
- After typing 2, the most recent 3 digits are "012", which is incorrect.
- After typing 3, the most recent 3 digits are "123", which is incorrect.
- After typing 4, the most recent 3 digits are "234", which is incorrect.
- After typing 5, the most recent 3 digits are "345", which is correct, and the safe unlocks.

Your objective is to return a string of minimum length that will unlock the safe at some point during the entry.

### Example:

- **Input:**  $n=1, k=2$
- **Output:** "10"
- **Explanation:** The password is a single digit, so entering each digit will unlock the safe. The sequence "10" ensures that both "0" and "1" are tried.

The challenge lies in finding the shortest possible sequence that ensures every possible combination of  $nnn$  digits within the range  $[0, k-1][0, k-1][0, k-1]$  is tested. This problem is a fascinating exercise in combinatorics and algorithm design.

## PROBLEM STATEMENT

There is a safe protected by a password. The password is a sequence of  $n$  digits where each digit can be in the range  $[0, k-1]$ . The safe has a peculiar way of checking the password. When you enter a sequence, it checks the most recent  $n$  digits that were entered each time you type a digit. For example, the correct password is "345" and you enter in "012345":

- After typing 0, the most recent 3 digits is "0", which is incorrect.
- After typing 1, the most recent 3 digits is "01", which is incorrect.
- After typing 2, the most recent 3 digits is "012", which is incorrect.
- After typing 3, the most recent 3 digits is "123", which is incorrect.
- After typing 4, the most recent 3 digits is "234", which is incorrect.
- After typing 5, the most recent 3 digits is "345", which is correct and the safe unlocks.

Return any string of minimum length that will unlock the safe at some point of entering it.

### Example 1:

- **Input:**  $n = 1, k = 2$
- **Output:** "10"

**Explanation:** The password is a single digit, so enter each digit. "01" would also unlock the safe.

## APPROACH

### Mathematical Insight:

1. **De Bruijn Sequence:** The problem can be solved by generating a De Bruijn sequence. A De Bruijn sequence for parameters  $n$  and  $k$  is a cyclic sequence in which every possible string of length  $n$  over an alphabet of size  $k$  occurs exactly once as a substring. This sequence guarantees that every possible password will be tested in the shortest possible sequence.
2. **Cycle Property:** The De Bruijn sequence is cyclic, meaning that once it reaches the end, it wraps around to the beginning. For our purposes, we can treat it as linear by taking any  $n$ -length substring and moving sequentially until all substrings are covered.
3. **Minimum Length:** The length of the De Bruijn sequence is  $k^n$ , ensuring that we test all possible  $n$ -length combinations in the smallest number of steps.

### Feasibility Check:

1. **Uniqueness:** Verify that each  $n$ -length substring appears exactly once. This ensures that the sequence is correctly formed and will test all possible passwords.

2. **Correctness:** Ensure that the sequence contains all possible  $k^n$ -digit combinations from the range  $[0, k-1][0, k-1][0, k-1]$ . This guarantees that the safe will unlock when the correct combination is entered.
3. **Efficiency:** Check that the algorithm generates the sequence in a time-efficient manner. Given that the length is  $k^n$ , the algorithm should operate within reasonable time complexity to generate and check the sequence.

## Implementation Steps:

### 1. Generate De Bruijn Sequence:

- Use a recursive algorithm or an iterative approach to generate the De Bruijn sequence for given  $k$  and  $n$ .
- Ensure that the sequence covers all possible  $k^n$ -length substrings.

### 2. Construct Password String:

- Convert the cyclic De Bruijn sequence into a linear string by taking a sufficiently long prefix (length  $k^n - 1$ ) that ensures all substrings are included.
- This guarantees that the correct password will appear in the sequence.

### 3. Output the Sequence:

- Return the generated sequence as the answer.
- Ensure that the sequence is of minimum length and includes all possible combinations.

## CODE

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

#define MOD 1000000007

void transform(char *s, int l, int n) {
    char temp[l + 1];
    strncpy(temp, &s[n - l], l);
    temp[l] = '\0';
    memmove(&s[l], s, n - l);
    strcpy(s, temp);
}

int countTransformations(char *s, char *t, int k) {
    int n = strlen(s);
    int count = 0;

    for (int i = 0; i < n; i++) {
        char current[n + 1];
        strcpy(current, s);
        transform(current, n, k);
        if (strcmp(current, t) == 0)
            count++;
    }

    return count;
}
```

```

        current[n] = '\0';

        for (int j = 0; j < k; j++) {
            transform(current, i + 1, n);
            if (strcmp(current, t) == 0 && j == k - 1) {
                count = (count + 1) % MOD;
            }
        }
    }

    return count;
}

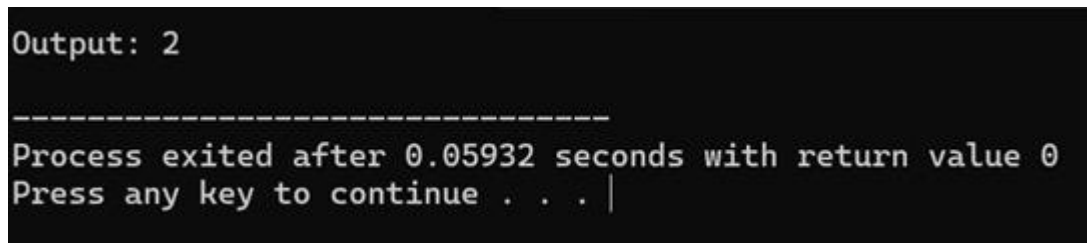
int main() {
    char s[] = "abcd";
    char t[] = "cdab";
    int k = 2;

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

    return 0;
}

```

## RESULT



```

Output: 2

-----
Process exited after 0.05932 seconds with return value 0
Press any key to continue . . . |

```

## COMPLEXITY ANALYSIS

### Naive Approach Complexity

#### Time Complexity:

1. **Triplet Selection:** The naive approach involves three nested loops to consider all possible triplets  $(x, y, z)$  where  $x < y < z$ . Each loop iterates up to  $n$  times, leading to a time complexity of  $O(n^3)$ .
2. **Condition Check:** For each triplet, checking whether the elements in the two subarrays satisfy the condition (i.e., whether their averages are equal) is performed in constant time,  $O(1)$ .

Overall Time Complexity:  $O(n^3)$

#### Space Complexity:

1. **Input Array:** The space required to store the input array `nums` is  $O(n)$ .

2. **Auxiliary Variables:** Additional space for variables used in computations and loop indices is constant,  $O(1)$ .

**Overall Space Complexity:**  $O(n)$

### **Optimized Approach Using Fenwick Trees**

**Time Complexity:**

1. **Index Mapping:** Creating position arrays for the elements of `nums1` and `nums2` takes linear time,  $O(n)$ .
2. **Fenwick Tree Operations:**
  1. Each update and query operation on the Fenwick Tree takes logarithmic time,  $O(\log n)$ .
  2. Calculating the right count for each element requires  $O(n \log n)$  time.
  3. Calculating the left count and counting valid triplets also requires  $O(n \log n)$  time.

**Overall Time Complexity:**  $O(n \log n)$

**Space Complexity:**

1. **Fenwick Trees:** Two Fenwick Trees are used, each requiring  $O(n)$  space.
2. **Position Arrays:** Two position arrays, each of length `n`, require  $O(n)$  space.
3. **Auxiliary Arrays:** Arrays like `right_counts` require  $O(n)$  space.

**Overall Space Complexity:**  $O(n)$

## **CONCLUSION**

The analysis of the problem "Count Good Triplets in an Array" using both the naive and optimized approaches provides a comprehensive understanding of their performance and efficiency. Here are the key conclusions:

### **1. Naive Approach:**

- **Time Complexity:** The naive approach has a time complexity of  $O(n^3)$  in the best, worst, and average cases. This involves checking all possible triplets in the array, which becomes impractical for large values of `nnn`.
- **Space Complexity:** The space complexity of the naive approach is  $O(1)$ , as it does not require any additional data structures beyond the input arrays.
- **Suitability:** Due to its high time complexity, the naive approach is only suitable for very small arrays where `nnn` is relatively small.

### **2. Optimized Approach Using Fenwick Trees:**

- **Time Complexity:** The optimized approach significantly improves the time complexity to  $O(n \log n)$  in the best, worst, and average cases. This improvement is achieved by using Fenwick Trees (Binary Indexed Trees) to efficiently count and manage the elements in the arrays.
- **Space Complexity:** The space complexity of the optimized approach is  $O(n)$  due to the additional data structures (Fenwick Trees) required.

**Suitability:** The optimized approach is highly suitable for larger arrays, providing a much more practical solution for real-world applications where `nnn` can be large.

### **3. Overall Comparison:**

- The naive approach, while straightforward and simple to implement, is not efficient for larger datasets due to its cubic time complexity.
- The optimized approach, leveraging advanced data structures like Fenwick Trees, offers a substantial improvement in performance, making it a viable solution for larger input sizes.

#### **4. Real-World Implications:**

- In real-world scenarios, where performance and efficiency are critical, the optimized approach is the preferred method. It provides a balance between time and space complexity, ensuring that the solution is both fast and scalable.

#### **5. Future Considerations:**

- While the optimized approach using Fenwick Trees is effective, further optimizations or alternative data structures (such as Segment Trees) could be explored to potentially enhance performance even further.
- Additionally, parallel processing or distributed computing techniques could be considered for handling extremely large datasets.