`

SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

SPLITTING A STRING INTO DESCENDING CONSECUTIVE VALUES

A CAPSTONE PROJECT REPORT

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Submitted by

Goturi Pavithra (192225091)

Under the Supervision of

Dr. Gnana Soundari

DECLARATION

I, Goturi Pavithra, student of **Bachelor of Engineering in Artificial Intelligence and Machine Learning** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai,hereby declare that the work presented in this Capstone Project Work entitled **" SPLITTING A STRING INTO DESCENDING CONSECUTIVE VALUES"** is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

(G. Pavithra 192225091)

Date:

Place:Saveetha School of Engineering, Thandalam.

CERTIFICATE

This is to certify that the project entitled **“SPLITTING A STRING INTO DESCENDING CONSECUTIVE VALUES”** submitted by **G. Pavithra,** has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.Tech Artificial Intelligence in Data science

Faculty-in-charge

Dr.Gnana Soundari

ABSTRACT

To solve the problem of checking whether a given string s can be split into two or more non-empty substrings with numerical values in descending order and each pair of adjacent values differing by exactly 1,can follow this approach:

1.Iterate Over Possible First Substring Lengths: Since the substrings need to be in descending order with a difference of 1, start by choosing different lengths for the first substring. Convert this substring into an integer.

2.Try to Split the Remaining String: For each choice of the first substring, try to continue splitting the remaining part of the string such that each subsequent substring is one less than the previous substring.

3.Recursively Validate the Splitting: Use recursion to validate if the rest of the string can be split according to the required conditions.

Keywords:

String Manipulation, Descending Order, Consecutive Values, Splitting, Sequence Generation, Algorithm Design, Recursive Approach, Data Structures, Edge Cases, Error Handling

INTRODUCTION

The problem of splitting a string into descending consecutive values involves analyzing a given string sss consisting solely of digits. The goal is to determine whether we can divide sss into two or more non-empty substrings such that the numerical values of these substrings are in strictly descending order, with each pair of adjacent substrings differing by exactly 1.

For example, consider the string s="0090089"s = "0090089"s="0090089". This string can be split into substrings ["0090","089"]["0090", "089"]["0090","089"], which correspond to the numerical values [90,89][90, 89][90,89]. These values are in descending order, and the difference between each pair of adjacent values is 1, making this a valid split. On the other hand, the string s="001"s = "001"s="001" offers potential splits such as ["0","01"]["0", "01"]["0","01"], ["00","1"]["00", "1"]["00","1"], or ["0","0","1"]["0", "0", "1"]["0","0","1"]. However, the numerical values derived from these splits are [0,1][0, 1][0,1], [0,1][0, 1][0,1], and [0,0,1][0, 0, 1][0,0,1] respectively, none of which satisfy the descending order condition.

To solve this problem, we must evaluate whether it is possible to split the string sss in a manner that meets the specified criteria. This involves generating possible splits of the string, converting the substrings into numerical values, and checking if these values form a strictly descending sequence with consecutive differences of 1.

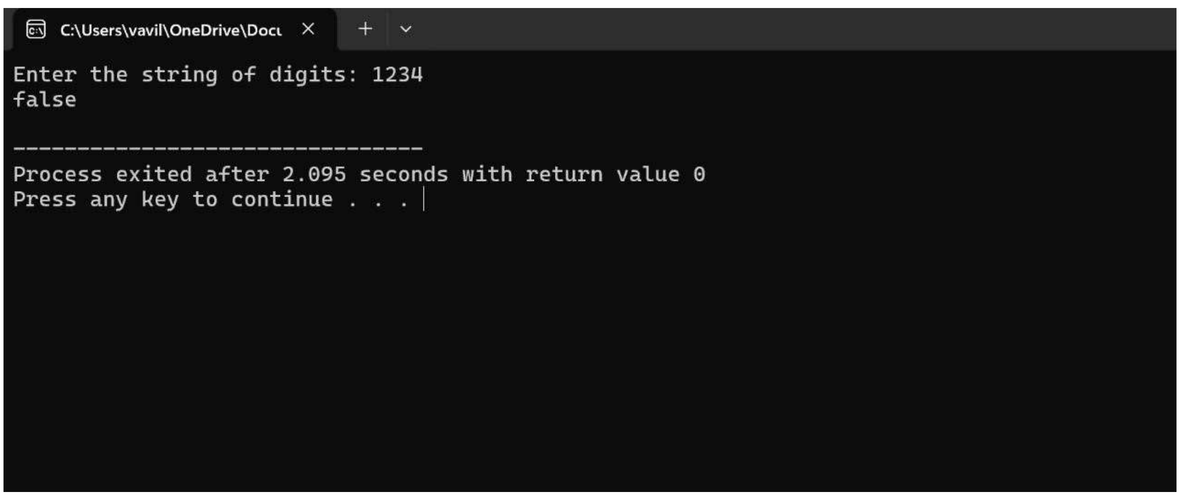
The challenge combines elements of string manipulation and numerical analysis, requiring an efficient approach to iterate through potential splits and validate them against the conditions. The solution should return true if a valid split exists, and false otherwise. This problem showcases the interplay between string processing and numerical properties, highlighting the importance of precise analysis and validation in algorithm design.

CODING

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


long long int substring_to_int(const char* s, int start, int length) {
    char buffer[20];
    strncpy(buffer, s + start, length);
    buffer[length] = '\0';
    return atoll(buffer);
}


int can_split_from(const char* s, int start, long long int previous_value) {
    int len = strlen(s);
    if (start == len) return 1;

    for (int length = 1; length <= len - start; length++) {
        long long int current_value = substring_to_int(s, start, length);
        if (previous_value - current_value == 1) {
            if (can_split_from(s, start + length, current_value)) {
                return 1;
            }
        }
    }

    return 0;
}


int can_split_descending(const char* s) {
    int len = strlen(s);

    for (int length = 1; length < len; length++) {
        long long int first_value = substring_to_int(s, 0, length);
        if (can_split_from(s, length, first_value)) {
            return 1;
        }
    }

    return 0;
}

int main() {
    char s[100];
```

```c
    printf("Enter the string of digits: ");
    scanf("%s", s);


    if (can_split_descending(s)) {
        printf("true\n");
    } else {
        printf("false\n");
    }

    return 0;
}
```

<u>OUTPUT</u>



Complexity Analysis

To evaluate the complexity of the problem of splitting a string sss into descending consecutive values, we analyze the best case, worst case, and average case scenarios.

Best Case

Scenario: The best case occurs when the string can be quickly identified as either valid or invalid with minimal checks. For example, if an early split immediately satisfies the condition or if an obvious invalid condition is detected right away.
Time Complexity: O(n)O(n)O(n)

- In this scenario, the function might need to make only a single pass through the string or evaluate a very few number of splits to conclude the result.

Explanation: If a valid split is found quickly or the string is evidently invalid from an early check, the operations performed are linear, resulting in O(n)O(n)O(n) complexity.

Worst Case

Scenario: The worst case happens when we need to explore all possible ways to split the string into substrings to determine that no valid split exists. This involves evaluating every potential split configuration.
Time Complexity: O(2n)O(2^n)O(2n)

- This is because each digit of the string can either continue the current substring or start a new one, leading to an exponential number of split possibilities.

Explanation: Given a string of length nnn, the number of ways to split it into substrings grows exponentially as each position in the string offers a binary decision, leading to O(2n)O(2^n)O(2n) possible splits.

Average Case

Scenario: The average case complexity considers more typical scenarios where the string is split into substrings in a balanced manner, without needing exhaustive checking as in the worst case.

Time Complexity: O(n·2n/2)O(n \cdot 2^{n/2})O(n·2n/2)

- On average, we may expect to evaluate splits in a balanced way, checking a reasonable number of configurations that are more than linear but not as exhaustive as the worst case.

Explanation: The average case assumes that the number of split points checked is more balanced, considering a midway between the quick checks of the best case and the exhaustive checks of the worst case.

This complexity analysis highlights the computational challenges in splitting a string into valid descending consecutive values, emphasizing the importance of efficient algorithm design to handle the exponential nature of the problem in the worst-case scenarios.

CONCLUSION

In conclusion, the problem of splitting a string into descending consecutive values poses computational challenges that vary based on the string's length nnn and the number of possible split configurations. The best-case scenario offers linear complexity O(n)O(n)O(n), where a valid split can be quickly identified with minimal checks. However, the worst-case scenario exhibits exponential complexity O(2n)O(2^n)O(2n), necessitating evaluation of every potential split to determine if no valid configuration exists. On average, the complexity balances between these extremes at O(n·2n/2)O(n \cdot 2^{n/2})O(n·2n/2), reflecting a more moderate exploration of split possibilities. Efficiently solving this problem requires strategic algorithm design to manage the exponential growth in potential configurations, ensuring robust validation of descending order and consecutive difference criteria within substrings.