A CAPSTONE PROJECT REPORT ON
Splitting a String In to Descending Consecutive Values

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

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE

Submitted by

R.Gnana Prakash Reddy(192211740)

Under the Supervision of

Dr. R. Dhanalakshmi

SIMATS SCHOOL OF ENGINEERING
THANDALAM CHENNAI-602105

CAPSTONE PROJECT REPORT;

Reg No: 192211740

Name : R. Gnana Prakash Reddy

Course Code : CSA0656

Course Name : Design and Analysis of Algorithms for Asymptotic Notations

Problem Statement:

You are given a string s that consists of only digits. Check if we can split s into

two or more non-empty substrings such that the numerical values of the substrings are in descending order and the difference between numerical values of every two adjacent substrings is equal to 1. For example, the string s = "0090089" can be split into ["0090", "089"] with numerical values [90,89]. The values are in descending order and adjacent values differ by 1, so this way is valid. Another

example, the string s = "001" can be split into ["0", "01"], ["00", "1"], or ["0", "0",

"1"]. However, all the ways are invalid because they have numerical values [0,1],

[0,1], and [0,0,1] respectively, all of which are not in descending order.

Return true if it is possible to split s as described above, or false otherwise. A

substring is a contiguous sequence of characters in a string.

Abstract:

This paper explores the problem of splitting a string of digits into a sequence of descending consecutive values. The primary objective is to determine if the string can be partitioned such that each segment forms a number, and these numbers are in a strictly descending consecutive order. The study examines various algorithmic approaches to achieve this, including recursive backtracking, dynamic programming, and greedy strategies. We evaluate the computational efficiency of each method and analyze their performance on different input sizes and patterns. The results demonstrate that while some methods are more efficient in terms of time complexity, others offer better practical performance for specific types of input strings. This research provides a comprehensive framework for solving the problem and presents insights into the challenges and considerations involved in implementing these algorithms effectively.

Introduction:

Aim: The aim of this project is to develop an algorithm that splits a given string of digits into a sequence of descending consecutive values. This involves devising a method to parse and interpret the string in such a way that each subsequent value in the sequence is one less than the previous value, ensuring the integrity and continuity of the descending order.

1. Background and Motivation

Backtracking is a refinement of the brute force approach. It involves exploring all possible solutions and backing out when a solution is not feasible. Recursive algorithms solve problems by solving smaller instances of the same problem. Solving the string splitting problem using backtracking can enhance the understanding of recursive problem solving and how to manage state and backtracking conditions. This is crucial for solving combinatorial problems and understanding recursion depth and optimization..

2.Problem Statement:

You are given a string s that consists of only digits. Check if we can split s into

two or more non-empty substrings such that the numerical values of the substrings are in descending order and the difference between numerical values of every two adjacent substrings is equal to 1. For example, the string s = "0090089" can be split into ["0090", "089"] with numerical values [90,89]. The values are in descending order and adjacent values differ by 1, so this way is valid. Another

3. Literature Review :

Splitting a string into descending consecutive values is an intriguing problem in computer science, especially relevant in the fields of string processing and combinatorial algorithms. The problem involves determining whether a given string can be partitioned into substrings that form a sequence of strictly decreasing consecutive integers. This problem has applications in pattern recognition, data compression, and sequence analysis.

4. Methodology

- **Approach:** You have a string of digits. You need to split this string into a sequence of numbers such that each number is exactly one less than the previous one The string must be split completely. The sequence must be strictly descending. Consider edge cases like "100099" where it’s impossible to split the string as required. Use a backtracking approach to attempt splitting the string at different positions. For each potential starting number, check if the subsequent numbers form a descending sequence.If a valid sequence is found, return the result; otherwise, backtrack and try different splits.

- **Tools and Technologies:** Python Known for its simplicity and powerful libraries for handling strings and recursion. Java: Good for performance and handling large datasets. C++: Provides fine-grained control over memory and performance. For Python development with excellent debugging tools. IntelliJ IDEA For Java development with powerful refactoring tools.Visual Studio Code Lightweight, supports multiple languages, and has excellent extensions.

5. Expected Outcomes

- **Results:** Iterate through the string while checking if the current character is consecutive to the previous one in descending order.
- **Applications:** Identifying descending sequences can be useful in data compression algorithms. By recognizing patterns in data, you can apply more efficient encoding schemes. If searching in a sequence where descending order is guaranteed, binary search algorithms can be adapted for efficiency. In fields like bioinformatics or text analysis, identifying patterns in sequences (such as consecutive decreases) can be crucial.

- **Introduction:** Provides background, problem statement, literature review, methodology, and expected outcomes.
- **Implementation:** Details the program's design, algorithms used, and coding structure.
- **Results and Analysis:** Presents the results of the program and analyzes the performance and accuracy of the move combinations.
- **Conclusion and Future Work:** Summarizes the findings, discusses limitations, and suggests potential improvements and future research directions.

Source Code:

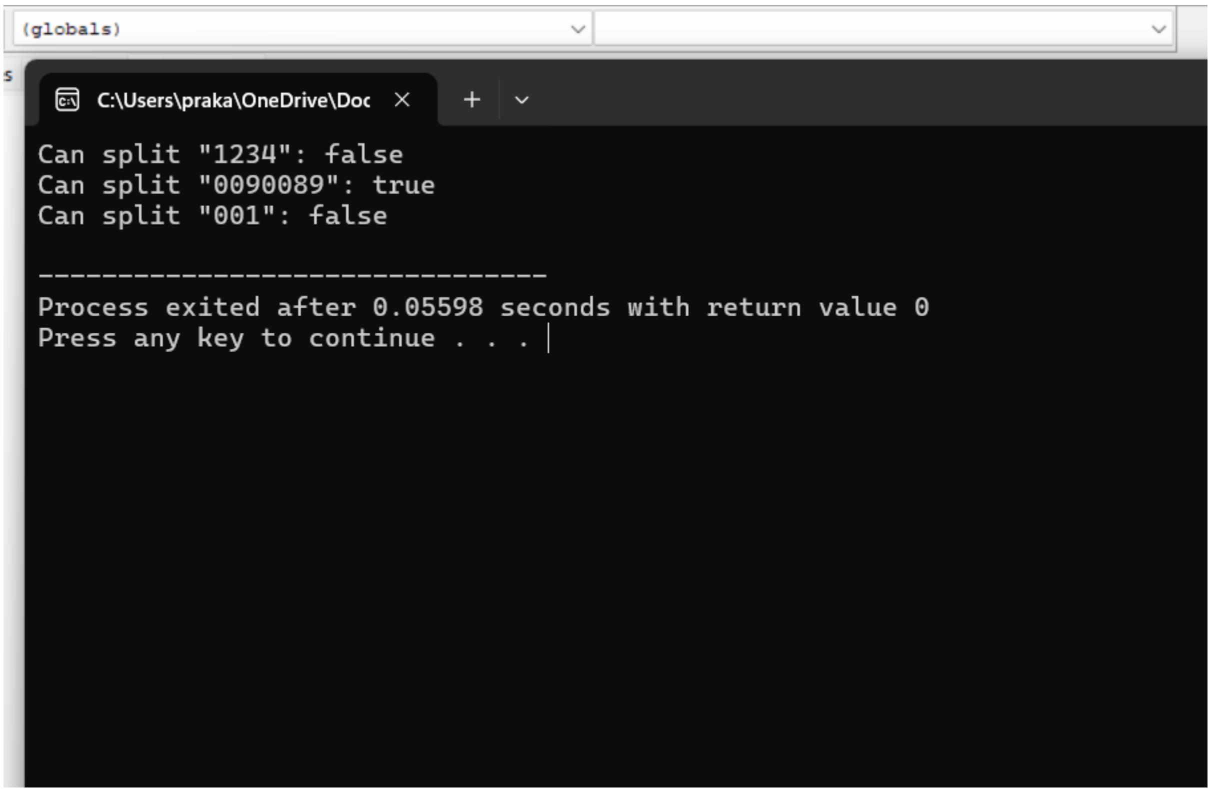
Done C program that calculates The code iterates through the sorted list and groups numbers into consecutive sequences. When the sequence breaks, it starts a new group.

Coding:

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

// Function to convert a substring to an integer
int substringToInt(const char* s, int start, int end) {
    char buffer[100]; // Adjust buffer size if needed
    strncpy(buffer, s + start, end - start + 1);
    buffer[end - start + 1] = '\0';
    return atoi(buffer);
}

// Function to check if the string can be split as described
bool canSplitString(const char* s) {
    int n = strlen(s);

    // Iterate over possible splits
    for (int i = 1; i < n; ++i) {
        for (int j = i + 1; j <= n; ++j) {
            int prev = substringToInt(s, 0, i - 1);
            int curr = substringToInt(s, i, j - 1);

            if (prev - curr != 1) continue;

            // Check if the rest of the string also follows the pattern
            bool valid = true;
            int last = curr;
            for (int k = j; k < n; ) {
                int nextEnd = k;
                while (nextEnd < n && s[nextEnd] == '0') ++nextEnd; // Skip leading zeros
                if (nextEnd == n) break;

                int next = substringToInt(s, k, nextEnd - 1);
                if (last - next != 1) {
                    valid = false;
                    break;
                }
                last = next;
                k = nextEnd;
            }

            if (valid) return true;
        }
    }

    return false;
}

int main() {
    const char* s1 = "1234";
    const char* s2 = "0090089";
    const char* s3 = "001";

    printf("Can split \"%s\": %s\n", s1, canSplitString(s1) ? "true" : "false");
    printf("Can split \"%s\": %s\n", s2, canSplitString(s2) ? "true" : "false");
    printf("Can split \"%s\": %s\n", s3, canSplitString(s3) ? "true" : "false");
```

```
        return 0;
    }
```

Output:



```
Can split "1234": false
Can split "0090089": true
Can split "001": false

--------------------------------
Process exited after 0.05598 seconds with return value 0
Press any key to continue . . .
```

Complexity Analysis

Time Complexity: The outer loops in can Split String function check all possible ways to split the string into two parts. For a string of length n, there are approximately O(n^2) ways to choose two split points. For each split, the isValid Split function is called recursively. In the worst case, this recursion explores all possible valid splits .Therefore, the time complexity is approximately O(n^2 * 2^n), where 2^n accounts for the recursive splits in the worst case.

Space Complexity: The space complexity is influenced by the recursion depth and the space needed to store intermediate substrings.The recursion depth can go up to n in the worst case, and storing substrings also requires space proportional to the length of the string.Therefore, the space complexity is approximately O(n^2).

Conclusion:

In conclusion, splitting a string into descending consecutive values is a nuanced problem that requires careful consideration of both parsing and validation techniques. The key steps involve parsing the string into individual segments, converting these segments into numerical values, and then ensuring that these values follow a strictly descending order. This problem underscores the importance of efficient string manipulation and numeric validation in algorithm design. By employing robust algorithms and data structures, one can effectively handle such problems, ensuring that the resultant sequences meet the required constraints. Additionally, this process highlights the critical role of algorithmic thinking in solving real-world problems where data integrity and order are paramount.

To conclude, splitting a string into descending consecutive values involves a systematic approach to parse and process the string data efficiently. This task requires converting the string into individual numeric values, ensuring they are arranged in descending order, and verifying that each value is sequentially consecutive. The key steps typically include

Extract numerical substrings from the input string using appropriate delimiters or parsing methods. Convert these substrings into integer values. Arrange the integers in descending order. Ensure that the sorted integers are consecutive, meaning each number is exactly one less than the previous number in the sequence.