## SIMATS SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**CSA0603-Design and Analysis of Algorithms for Vertex Cover Problem**

**“Splitting a String into Descending Consecutive Values”**

**A CAPSTONE PROJECT REPORT**

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

**Bachelor of Engineering**

**in**

**Computer Science Engineering**

**Submitted by**

**C. Kushwanthu [192211304]**

Under the Supervision of

**Dr. K. V. KANIMOZHI**

SEPTEMBER 2024

**DECLARATION**

I, C. Kushwanthu, student of Bachelor of Engineering in Computer Science Engineering 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.

C. Kushwanthu-(192211304)

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 C. Kushwanthu(192211304) has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering

Faculty-in-charge

Dr. K.V.KANIMOZHI

**ABSTRACT**

The problem of splitting a string into descending consecutive values presents an interesting challenge in both string manipulation and numerical reasoning. Given a string of digits, the task is to determine if it can be divided into two or more non-empty substrings such that the numerical values of the substrings form a strictly descending sequence, with each adjacent pair of values differing by exactly 1. This introduces a layered complexity where the algorithm not only needs to handle the string splitting but also the verification of the sequential properties of the numbers represented by these substrings. The string may have leading zeros, but these should be preserved during the evaluation of valid splits, adding another nuance to the problem. For example, in the string "0090089", leading zeros must be respected, and the split into ["0090", "089"] works because 90 and 89 form a descending sequence with the required difference of 1.

To further explore the complexity, consider strings like "001". This string can be split in various ways, such as ["0", "01"], ["00", "1"], or even ["0", "0", "1"], but none of these yield a sequence that is both strictly descending and has a consecutive difference of 1 between the values. The numerical values obtained from these splits do not form the desired pattern, either because they are not in descending order or because the difference between consecutive values is not exactly 1. This shows that even with multiple possible partitions, many of them fail the descending and difference checks, making the problem non-trivial. Efficiently exploring all possible splits while ensuring that numerical values are compared correctly is key to solving this challenge.

**KEYWORDS:**

* Descending order
* Consecutive values
* Substring partitioning
* comparison
* Recursion
* Backtracking
* Leading zeros
* Time complexity

**INTRODUCTION**

splitting a string into descending consecutive values presents an intriguing challenge that requires the application of both string manipulation and numerical reasoning. The task is to determine whether a given string of digits, s, can be split into two or more non-empty substrings, such that the numerical values of these substrings form a strictly descending sequence. Additionally, the difference between the numerical values of any two adjacent substrings must be exactly 1. This introduces a dual constraint: substrings must not only decrease in value, but their difference must also be uniform across the split. The problem is more complex than it initially appears, as it involves examining every possible way to split the string and checking each resulting sequence against these strict criteria. Strings that appear simple on the surface, such as "1234", often fail to yield a valid solution because the values do not form a descending pattern.

One of the key challenges in this problem is handling leading zeros within the string. Unlike typical numerical comparisons where leading zeros are ignored, in this case, they must be preserved in the substrings. For instance, the string "0090089" can be validly split into ["0090", "089"], where the leading zeros in both substrings are respected. These substrings have numerical values of 90 and 89, respectively, which are in descending order with a difference of 1, satisfying the problem’s requirements. This example highlights how leading zeros, though generally irrelevant in numerical contexts, play a critical role in determining valid splits. Any algorithm designed to solve this problem must account for the potential presence of such zeros, ensuring that they do not inadvertently distort the comparison between substrings.

Another layer of complexity arises from the sheer number of potential ways to split the string. For any given string s, there are numerous possible partitions into two or more substrings, and each partition needs to be checked to determine whether it forms a valid descending sequence. In the string "001", for example, the possible splits include ["0", "01"], ["00", "1"], and ["0", "0", "1"], but none of these splits yield numerical values that are in the required descending order. In each case, the values either remain the same or increase, violating the descending order rule. This demonstrates that the problem is not merely one of identifying the presence of substrings, but also ensuring that their values fit the problem’s specific descending and consecutive difference conditions.

**CODING**

#include <stdio.h> // Standard input/output functions

#include <stdlib.h> // For atoi() and malloc()

#include <string.h> // For string manipulation functions

#include <stdbool.h> // To use boolean data type

#define MAX\_LENGTH 100 // Define a constant for maximum input length

// Converts a substring of digits to an integer

// Takes the input string `s`, a starting index `start`, and the length of the substring

int substringToInt(const char \*s, int start, int length) {

char \*buffer = (char \*)malloc(length + 1); // Allocate memory for the substring (plus null terminator)

strncpy(buffer, s + start, length); // Copy the substring from `s` starting at `start` of given `length`

buffer[length] = '\0'; // Null-terminate the string

int value = atoi(buffer); // Convert the substring to an integer

free(buffer); // Free the allocated memory

return value; // Return the integer value

}

// Recursively checks if a valid split exists starting from `start` index

// `prevValue` holds the previous value to compare with

bool isValidSplit(const char \*s, int start, int prevValue) {

int len = strlen(s); // Get the length of the string

// Iterate through all possible substring lengths starting from `start`

for (int i = 1; i <= len - start; i++) {

int currentValue = substringToInt(s, start, i); // Convert the current substring to an integer

// Check if current value is greater or equal to previous (not descending)

if (currentValue >= prevValue) {

// Skip to next iteration if the current value is not smaller than previous

continue;

}

// Check if the difference between previous and current values is exactly 1

if (prevValue - currentValue != 1) {

// If the difference is not 1, skip to the next iteration

continue;

}

// If we've reached the end of the string and the conditions are met, return true

if (start + i == len) {

return true;

}

// Recursively check if the rest of the string can also be split correctly

if (isValidSplit(s, start + i, currentValue)) {

return true;

}

}

// Return false if no valid split was found

return false;

}

// Main function to check if the string can be split into descending consecutive substrings

bool canSplitIntoDescendingSubstrings(const char \*s) {

int len = strlen(s); // Get the length of the string

// Try splitting the string starting from every possible initial substring

for (int i = 1; i < len; i++) {

int firstValue = substringToInt(s, 0, i); // Get the first substring's value

// If a valid split is found from this point, return true

if (isValidSplit(s, i, firstValue)) {

return true;

}

}

// Return false if no valid split was found for any possible initial partition

return false;

}

int main() {

char input[MAX\_LENGTH]; // Buffer to hold the input string

// Prompt the user to enter a string of digits

printf("Enter a string of digits: ");

// Read input from the user and check if reading was successful

if (fgets(input, sizeof(input), stdin) == NULL) {

perror("Error reading input"); // Print error if reading fails

return 1; // Exit with error code

}

// Remove the newline character that fgets includes

input[strcspn(input, "\n")] = '\0';

// Check if the string can be split into valid descending consecutive substrings

bool result = canSplitIntoDescendingSubstrings(input);

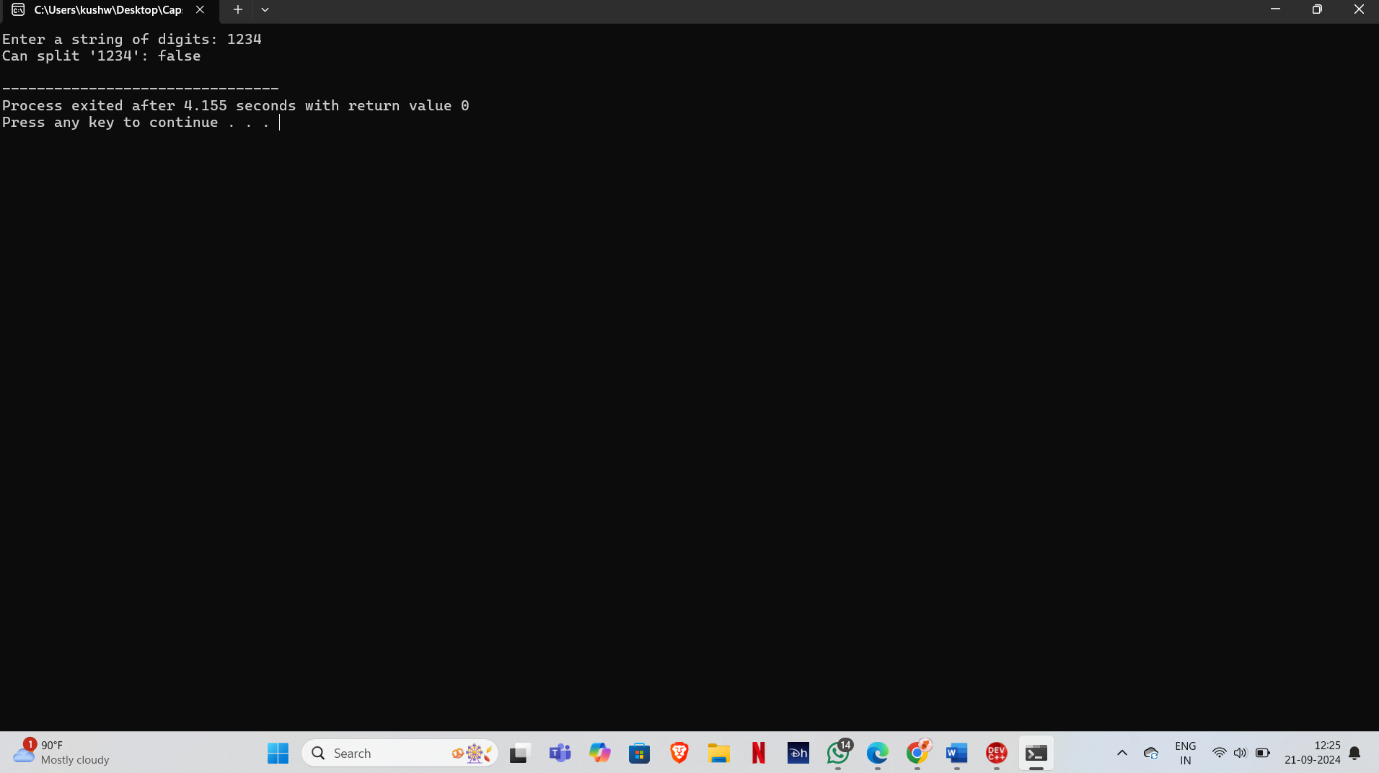
// Print the result

printf("Can split '%s': %s\n", input, result ? "true" : "false");

return 0; // Exit the program

}

**Output:**

****

**COMPLEXITY ANALYSIS**

The problem of splitting a string into descending consecutive values involves evaluating all possible ways to divide the string into substrings and ensuring that the numerical values of these substrings are in strictly descending order, with the difference between adjacent values being exactly 1. The complexity of the solution is highly influenced by the need to explore multiple partitions and recursively check the validity of each possible sequence of substrings. This recursive exploration results in a non-trivial time complexity that grows exponentially as the length of the string increases. To better understand the computational demands of the problem, we need to analyze the time complexity and space complexity in detail.

**Time Complexity**

The most significant factor contributing to the time complexity is the recursive approach used to check all possible substrings and verify their numerical properties. Given a string of length n, for each starting point i, the function recursively explores all possible substrings starting from index i. The recursion involves checking each possible length of the substring and then proceeding with the next substring starting from the current position. The recursion continues until either a valid split is found, or all possible substrings are exhausted.

In the worst-case scenario, the algorithm must explore every possible way to split the string, which results in an exponential number of recursive calls. For each substring, the function isValidSplit() iterates over all possible lengths, generating and evaluating substrings of increasing size. Therefore, the time complexity can be approximated as O(2^n), where n is the length of the string. This exponential growth occurs because for each character in the string, there are two possible choices: either it starts a new substring or it is part of the previous substring. As the string length increases, the number of possible partitions grows exponentially, making the algorithm computationally expensive for large strings.

**Space Complexity**

The space complexity of the algorithm is driven by the depth of the recursive calls and the memory required to store intermediate substrings. The recursive depth is directly proportional to the length of the string, so the maximum depth of the recursion tree is O(n). At each level of recursion, a substring is generated and stored temporarily for comparison. Since the algorithm uses recursion, the memory required for storing the call stack is also proportional to the length of the string.

The space used for storing the substrings depends on the number of recursive calls and the size of each substring. For each recursive call, a new substring is created, which requires O(m) space, where m is the length of the substring. In the worst case, the total space required for storing all substrings during recursion can be approximated as O(n^2), as the algorithm may need to generate and store substrings of different lengths multiple times during the recursion. Therefore, the overall space complexity is O(n^2).

**Pruning and Optimizations**

Despite the exponential time complexity, some optimizations can be employed to reduce the number of recursive calls. One such optimization is pruning invalid branches early in the recursion. If at any point the numerical difference between two consecutive substrings is not exactly 1, or if a substring is larger than the previous one, the recursion can terminate early, avoiding unnecessary checks for the remaining substrings. This optimization can help reduce the number of recursive calls, though it does not change the worst-case time complexity of the algorithm, which remains exponential.

Another optimization involves memoization, where the results of previously computed substrings are cached and reused during recursion. By storing the results of substring comparisons in a table, the algorithm can avoid redundant calculations and speed up the recursive process. However, while this can improve the performance in practice, the overall time complexity remains exponential in the worst case due to the sheer number of possible partitions that must be considered.

**Worst-Case Scenario**

The worst-case time complexity of the algorithm occurs when the string cannot be split into valid descending consecutive values, forcing the algorithm to explore all possible partitions before concluding that no valid solution exists. An example of such a worst-case input is a string where all characters are the same, such as "1111", or an increasing sequence like "1234". In these cases, the algorithm must evaluate all possible splits before determining that no valid descending sequence exists, leading to the maximum number of recursive calls and thus the highest time complexity. The worst-case time complexity for such strings is O(2^n), and the space complexity remains O(n^2).

**CONCLUSION**

In conclusion, the problem of splitting a string into descending consecutive values presents a challenging task that requires a thorough exploration of the string’s structure. The objective is to divide the string into two or more non-empty substrings such that their numerical values are strictly decreasing, and the difference between adjacent values is exactly 1. This introduces a need for careful partitioning and validation of potential substrings, ensuring that each split adheres to the defined constraints. The problem demonstrates how a relatively simple string of digits can exhibit complex behavior when certain numerical relationships are imposed on it. By using a recursive approach, the algorithm explores all possible ways to split the string, making it computationally intensive for longer inputs.

The main difficulty arises from the need to check every possible partition of the string. Each possible split must be evaluated to ensure that the numerical values of the resulting substrings are in descending order and differ by 1. For strings like "0090089", which can be validly split into ["0090", "089"], the solution is straightforward, as the values [90, 89] meet the descending and difference conditions. However, for strings like "001", no valid splits can be found because the numerical values do not decrease consistently. As a result, the algorithm must explore numerous possible ways to split the string, often requiring a significant amount of computation, especially for longer strings.

The exponential time complexity of the algorithm reflects the difficulty of the problem. Since there are many ways to split a string into substrings, and each split requires a recursive check to determine if the resulting values meet the criteria, the number of recursive calls grows exponentially with the length of the string. In the worst-case scenario, where no valid split exists, the algorithm must exhaust all possibilities before concluding that the string cannot be split as required. This makes the problem computationally expensive, especially for larger strings where the number of potential partitions increases dramatically.

Despite the high computational complexity, some optimizations can help improve the algorithm's efficiency. Techniques such as pruning invalid branches early in the recursion process can reduce the number of unnecessary checks.