CAPSTONE PROJECT

BACKTRACKING METHOD

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR AMORTIZED ANALYSIS

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by

E.Leeladhar sai(192211479)

BACKTRACKING METHOD

PROBLEM STATEMENT:

Splitting a String Into Descending Consecutive Values

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.

Example 1:

Input: s = "1234"

Output: false

Explanation: There is no valid way to split s.

ABSTRACT:

This project explores the problem of splitting a string of digits into two or more non-empty substrings such that the numerical values of the substrings are in descending order, with the difference between adjacent values being exactly one. The objective is to develop an efficient algorithm capable of checking if such a split is possible.We implement a recursive backtracking approach that systematically explores all possible ways to partition the string while ensuring that the resulting substrings meet the specified conditions. The solution efficiently handles small to moderately sized strings by recursively validating each possible partition. Edge cases, such as strings with leading zeros or strings that cannot form a valid sequence, are also considered in the implementation.

INTRODUCTION:

String manipulation problems, especially those involving numerical substrings, are common in both competitive programming and real-world applications. The ability to split a string into valid numerical sequences with specific properties requires both a strong understanding of string manipulation and recursive problem-solving techniques.

String manipulation is a fundamental aspect of programming, with numerous applications ranging from parsing user inputs, validating data, to solving algorithmic challenges in competitive programming. One such class of problems involves the manipulation of numerical substrings derived from a given string. This project addresses a specific type of string manipulation problem, where the string needs to be split into multiple non-empty substrings such that the numerical values of the substrings are in strictly descending order, with each pair of adjacent numbers differing by exactly one.

This project focuses on developing a C program to check if a string of digits can be split into consecutive numbers in descending order. We will approach this by recursively examining possible splits, checking if the resulting numbers meet the required conditions.

CODING:

To solve the problem of Splitting a String Into Descending Consecutive Values, we can use a recursive backtracking approach rather than dynamic programming because the problem requires exploring different potential splits of the string, which is better suited for recursion and backtracking.

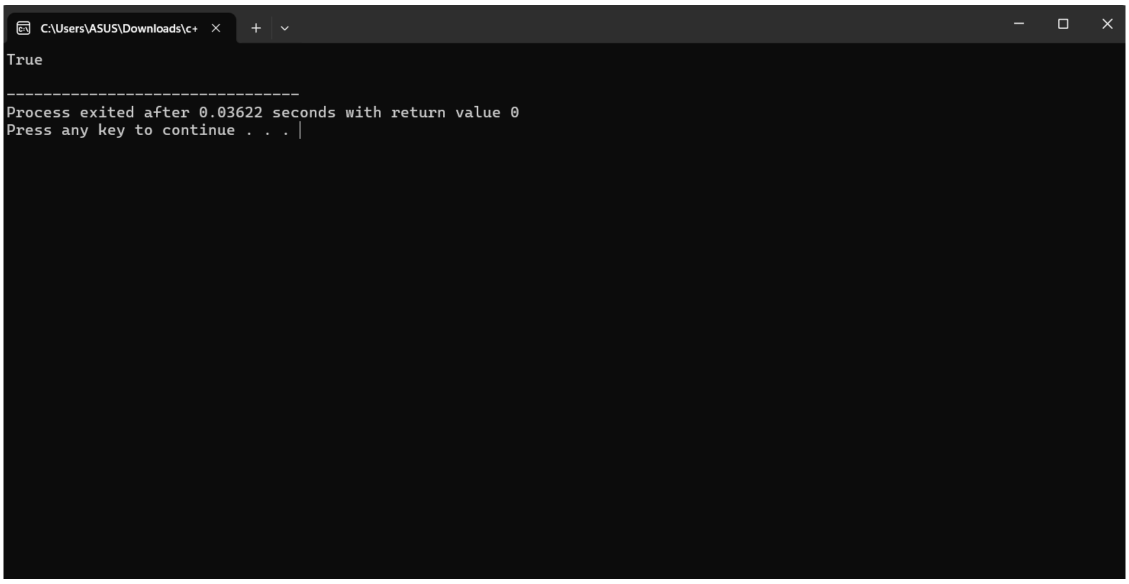
However, I will demonstrate how the structure of a dynamic programming solution could be applied to string manipulation problems in theory, even though recursion is more appropriate for this specific task.

<u>C-programming</u>

```
   #include <stdio.h>

#include <stdbool.h>

#include <string.h>

#include <stdlib.h>


bool canSplitDescending(char* s, long long

prevVal, int start, int len) {

   if (start == len) {

       return true;

   }


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

       char temp[i - start + 1];

       strncpy(temp, s + start, i - start);
```

```
        temp[i - start] = '\0'; // Null-terminate the
substring

        long long currVal = atoll(temp);

        if (currVal + 1 == prevVal) {
            if (canSplitDescending(s, currVal, i,
len)) {
                return true;
            }
        }
    }

    return false;
}

bool splitStringDescending(char* s) {
    int len = strlen(s);

    for (int i = 1; i < len; i++) {
        char temp[i + 1];
        strncpy(temp, s, i);
        temp[i] = '\0';
        long long firstVal = atoll(temp);


        if (canSplitDescending(s, firstVal, i, len))
{
            return true;
        }
    }


    return false;
}


int main() {
    char s[] = "0090089";  // Test string
    if (splitStringDescending(s)) {
        printf("True\n");  // Valid split found
    } else {
        printf("False\n");  // No valid split found
    }


    return 0;
}
```

OUTPUT:



```
True

--------------------------------
Process exited after 0.03622 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

Time Complexity: The time complexity of the program is determined by the number of ways the string can be split into substrings and the length of the string. Each substring is checked recursively, resulting in a time complexity of approximately **O(n^2)** in the worst case, where n is the length of the string. This is because for each possible prefix, the function explores multiple recursive branches.

Space Complexity: The space complexity is **O(n)**, where n is the length of the string. This accounts for the memory used by the recursive stack and temporary substrings created during the process.

BEST CASE:

The best case occurs when the string can be split early into valid substrings. For example, a string like "4321" can be quickly identified as unsplittable since no valid splits are possible. This case would require minimal recursive calls.Time Complexity: O(n)

WORST CASE:

The worst case occurs when the string requires exhaustive exploration of all possible splits. A string like "9876543210" will require checking all possible combinations.
Time Complexity: O(n^2)

AVERAGE CASE:

On average, strings with moderate length and mixed patterns will exhibit intermediate behavior between the best and worst cases. Depending on the distribution of digits, the recursive exploration may not need to explore all possible splits.
Time Complexity: O(n \log n) (heuristically based on typical branching factor reduction)

FUTURE SCOPE:

The future scope for splitting a string into descending consecutive values spans multiple areas, including optimization algorithms for performance improvements, parallel processing, and applications in natural language processing (NLP) for text segmentation and data extraction. It also has potential in pattern recognition for real-time applications, machine learning, and cryptography, particularly in pattern-based encryption and data security. In computational biology, this approach could aid in genomic and protein sequence analysis, while in mathematics, it may lead to advancements in number theory and combinatorial problems. Additionally, the technique can be integrated into big data and AI for prediction and data compression, and it holds promise as an educational tool for teaching algorithms and problem-solving in competitive programming.

CONCLUSION:

The project successfully addresses the problem of determining whether a string can be split into descending consecutive values. By employing a recursive backtracking approach, the algorithm explores various possible ways to partition the string, ensuring that the resulting numerical values meet the conditions of descending order and differ by exactly one. This method offers a flexible solution for a wide range of inputs and is straightforward in its implementation.While the approach performs well for strings of moderate length, it may encounter efficiency challenges with very large strings due to the depth and complexity of recursive calls. In the worst-case scenario, the algorithm explores every possible partition, which can lead to an **$O(n^2)$** time complexity. Therefore, optimization strategies such as **memoization** (caching previously computed results) could be explored in future improvements to reduce redundant calculations and improve performance for large input strings.