Application of Dynamic Programming For Efficient Text-Based Files Comparison

Rachel Gabriela Chen : 13521044
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13521044@std.stei.itb.ac.id

***Abstract*— This paper explores the application of dynamic programming for efficient comparison in text-based files comparison. Text-based files comparison is an important feature and can be used in version control system, plagiarism detection application, data analysis, and data management. Dynamic programming provides an effective approach to track and compare changes between different files. The proposed algorithm utilizes dynamic programming tables, subproblem decomposition, and optimal substructure to determine the longest common subsequence. Experimental evaluation demonstrates the superiority of the dynamic programming solution in terms of time complexity, making it a valuable approach for efficient text-based comparison. This paper highlights the practical implications of dynamic programming for text-based files comparison.**

Keywords—dynamic programming; text-based files; longest common subsequence

I. INTRODUCTION

Text-based file comparison plays a crucial role in various domains, including software development, document management, and data analysis. When dealing with large volumes of textual data, accurately identifying differences and similarities between files becomes a complex task. Efficient and precise file comparison is essential for tasks such as version control, plagiarism detection, content synchronization, and data integration. By employing advanced techniques and algorithms, text-based file comparison enables the automatic detection of changes, facilitating efficient data processing, decision-making, and ensuring data integrity. In this article, we will explore the challenges involved in text-based file comparison and delve into the various methodologies and approaches used to achieve accurate and efficient comparisons, ultimately enabling better management and analysis of textual data.

Dynamic programming is a powerful algorithmic approach that breaks down complex problems into smaller, overlapping subproblems, allowing for efficient computation of optimal solutions. Its application in version control can significantly improve the efficiency of file comparisons. By leveraging dynamic programming, we can determine the longest common subsequence between two files, which represents the unchanged or minimally changed portions. This approach reduces the computational complexity of file comparisons, enabling faster and more accurate results.

This paper explores the application of dynamic programming for efficient text-based files. Specifically, the paper focus on the longest common subsequence problem, which serves as a fundamental building block for file comparisons. This paper proposes an algorithm that utilizes dynamic programming tables, subproblem decomposition, and optimal substructure to identify the longest common subsequence between files. The paper's experimental evaluation demonstrates the superiority of the dynamic programming solution in terms of time complexity, highlighting its effectiveness for efficient text-based files comparison.

The efficient comparison of files is crucial for effective collaboration, conflict resolution, and maintaining a comprehensive revision history. The results of this study have practical implications for developers, technical writers, and other professionals who rely on version control systems. They can benefit from improved efficiency and accuracy in tracking and analyzing changes in text-based files.

II. THEORETICAL BASIS

A. *Text Based Files*

Text-based files are computer files that store data in plain text format, meaning they contain human-readable characters and can be edited with a simple text editor. These files are widely used for storing and exchanging information in various applications and systems.

Here are a few common types of text-based files:

1. Plain Text Files: These files contain unformatted text with no special styling or formatting. They typically have a ".txt" extension and can be opened and edited by any basic text editor.
2. Configuration Files: Many software applications use text-based configuration files to store settings and preferences. These files often have specific syntax and structure that the application understands and interprets.

3. Markup Languages: Markup languages like HTML (Hypertext Markup Language) and XML (eXtensible Markup Language) are text-based files that use specific tags and elements to define the structure and presentation of data. They are widely used for web development, document storage, and data interchange.
4. Programming Source Code: Source code files, such as those written in programming languages like Python, Java, or C++, are also text-based. They contain the instructions and commands that make up a computer program and can be edited with a code editor or an integrated development environment (IDE).
5. CSV Files: CSV (Comma-Separated Values) files store tabular data in plain text format, where each value is separated by a comma. They are commonly used for data storage, exchange, and analysis in spreadsheet programs and database systems.
6. Log Files: Applications and systems often generate log files that record events, activities, and errors. Log files are typically text-based and provide valuable information for troubleshooting and analysis.

The advantage of text-based files is their simplicity and interoperability. They can be easily opened, edited, and read by humans as well as various software applications across different platforms. Additionally, text-based files are often smaller in size compared to binary files, which can be beneficial for storage and transmission.

B. Dynamic Progamming

Dynamic programming is a problem-solving technique developed by Richard Bellman in the 1950s. Dynamic programming requires two important properties in a problem to be applicable: optimal substructure and overlapping subproblems. If a problem can be solved by combining optimal solutions to non-overlapping subproblems, it is considered a "divide and conquer" strategy, rather than dynamic programming.

Optimal substructure refers to the property where the solution to a given optimization problem can be obtained by combining optimal solutions to its subproblems. Typically, these optimal substructures are described using recursion. For instance, consider a graph G=(V,E) and the problem of finding the shortest path p from a vertex u to a vertex v. The shortest path p exhibits optimal substructure: if we take any intermediate vertex w on this path, we can split p into sub-paths p_1 from u to w and p_2 from w to v. These sub-paths are also the shortest paths between their respective vertices.

Dynamic programming takes into account overlapping sub-problems that might be solved repeatedly in a naïve recursive solution and solves each sub-problem in once. For example, consider the recursive formula in fibonacci sequence:

$$f(n) = f(n-1) + f(n-2)$$

with base case $f(0) = f(1) = 1$. Then $f(4) = f(3) + f(2)$ and $f(3) = f(2) . f(1)$. It is visible that $f(2)$ will be solved in the recursive sub-tress of both $f(4)$ and $f(3)$. Unlike the naïve approach that solves the same problem over and over, dynamic programming calculates the solution to $f(2)$ only once.

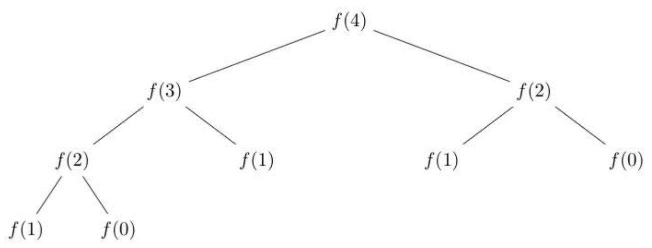


Figure 1. Fibonacci Recursion Tree (Source: math.stackexchange.com)

Dynamic programming can be categorized into two different approaches:

1. **Bottom-up (Tabulation) Approach**

   In the bottom-up approach, also known as tabulation, we start by solving the smallest subproblems and progressively build up to the larger problem. We create a table or array to store the solutions to these subproblems. The table is usually initialized with base cases or values that represent the simplest form of the problem.

   We then iteratively fill in the table, solving each subproblem once and storing its solution in the table. By using the solutions of previously solved subproblems, we can compute the solution for larger subproblems until we reach the final problem.

   This approach ensures that each subproblem is solved only once, and its solution is readily available when needed. It guarantees optimal time and space complexity by avoiding redundant computations.

2. **Top-down (Memoizaition) Approach**

   In the top-down approach, also known as memoization, we start with the original problem and recursively break it down into smaller subproblems. However, we optimize the approach by storing the solutions of already solved subproblems in a memoization table or cache. Before solving a subproblem, we first check if its solution exists in the cache. If so, we retrieve it; otherwise, we solve the subproblem and store its solution in the cache for future use.

   This approach utilizes the concept of memoization, where computed results are remembered and reused to avoid recomputation of the same subproblems. It helps reduce the overall time complexity by avoiding redundant calculations and focusing on unique subproblems.

   The top-down approach is often more intuitive and easier to implement recursively, as it follows the natural structure of the problem. However, it may require additional memory to store the cache for memoization.

Each technique has their own pros and cons:

Table 1. Top-Down and Bottom-Up comparison

Top-down	Bottom-up
Pros: • Interface errors can be more easily identified and isolated. • When errors occur at the top of the program, it provides advantages. • Early detection of design flaws allows for timely correction, as an initial functional module of the program is accessible.	Pros: • Creating test conditions is straightforward. • Observing test results is convenient. • It is particularly suitable when defects manifest at the lower sections of the program.
Cons: • Observing the test case output can be challenging. • Emphasizing the importance of writing stubs, as they determine the configuration of output parameters. • When stubs are located distant from the top-level module, selecting test cases and designing stubs becomes more complex.	Cons: • There is no representation of the working model once several modules have been constructed. • There is no existence of the program as an entity without the addition of the last module. • From a partially integrated system, test engineers cannot observe system-level functions. It can be possible only with the installation of the top-level test driver.

The main steps of solving a dynamic programming problem includes:

1. Determine whether the problem falls under the category of dynamic programming.
2. Choose a concise state expression with the minimum required parameters.
3. Establish the relationships between states and transitions.
4. Perform tabulation (or memoization) to compute and store the solutions.

In general, Dynamic Programming can be employed to solve a wide range of problems. These include situations where the goal is to maximize or minimize specific quantities, counting problems that involve determining the number of arrangements satisfying certain conditions, or probability problems. Some examples are:

1. Minimum cost path
2. Subset sum problem
3. Knapsack problem
4. Coin change problem
5. Longest common subsequence

C. *Longest Common Subsequence (LCS)*

The main description of longest common subsequence problem is as follows:

> Given two strings, S1 and S2 the task is to find the length of the longest subsequence present in both of the strings. (A subsequence of a string is a sequence that is formed by removing certain characters (potentially none) from the original string while maintaining the relative order of the remaining characters.

Input	**Output**

S1 = “AGGTAB”
S2 = “GXTAYB”
	LCS = “GTAB”

S1 = “ABCDGH”
S2 = “AEDFHR”
	LCS = “ADH”

LCS can be solved with recursive approach utilizing the following observation:

1. Let the input string be array of characters. S1[0…m-1] and S2[0…n-1] of lengths m and n respectively.
2. Let L(S1[0…[m-1], S2[0…n-1]) the length of the LCS of the two strings.
3. The recursive definition of L(S1[0…[m-1], S2[0…n-1]):
4. If the last characters of both strings match then L(S1[0…[m-1], S2[0…n-1]) = 1 + L(S1[0…[m-2], S2[0…n-2])
5. Else, L(S1[0…[m-1], S2[0…n-1]) = MAX (L(S1[0…[m-2], S2[0…n-1]), L(S1[0…[m-1], S2[0…n-2]))

The above expression in pseudocode:

```
function lcs(S1, S2, m, n):
    if m equals 0 or n equals 0:
        return 0
    else if S1[m-1] equals S2[n-1]:
        return 1 + lcs(X, Y, m-1, n-1)
    else
        return maximum of lcs(X, Y, m, n-1) and lcs(X, Y, m-1, n)
```

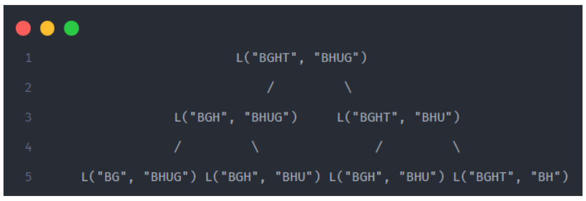
The recursive solution hold these properties:

1. **Optimal Substructure**

   L (S1[0…[m-1], S2[0…n-1]) is solved with the help of L (S1[0…[m-2], S2[0…n-2]) substructure.

2. **Overlapping Subproblems**

   Consider the strings S1 = "BGHT" and S2 = "BHUG". Using the recursive approach defined above, the recursion tree gained is as follows. It is visible that L("BGH", "BHU") is being calculated twice which shows a case of overlapping subproblems.



```
                      L("BGHT", "BHUG")
                          /        \
                 L("BGH", "BHUG")     L("BGHT", "BHU")
                 /        \               /        \
     L("BG", "BHUG") L("BGH", "BHU") L("BGH", "BHU") L("BGHT", "BH")
```

Figure 2. LCS Recursion Tree for "BGHT" and "BHUG"

The properties above are indicators that the longest common substring can be solved more efficiently with dynamic programming rather than recursive approach.

Longest Common Sequence (LCS) can become the basis to track changes and compare two distinct text-based files.

III. Application Of Dynamic Programming For Efficient Text-Based Files Comparison

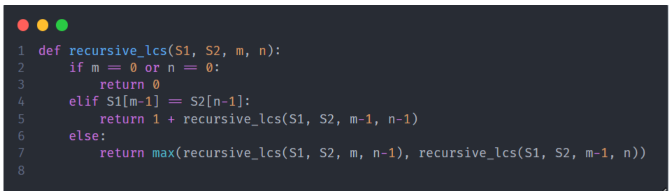
A. *Dynamic Programming Approach to Longest Common Subsequence Problem*

The steps to solving Longest Common Subsequence with dynamic programming are as follows:

1. Let S1 and S2 be the strings to compare, each with length m and n.
2. Create a 2D array dp[][] with m rows and n columns. The rows represent the indices of S1, meanwhile the columns represent the indices of S2.
3. The first row and column are initialized to 0.
4. Iterate the row starting with the first row with i = 1, for each row, iterate all the columns from j = 1 to n. If S1[i-1] equals S2[j-1], set the current element to the value of dp[i-1][j-1] +1. Elsem set the current element to the maximum between dp[i-1][j] and dp[i][j-1].
5. The last element in the dp array is the length of the LCS.

B. *Comparing Between Dynamic Programming and Recursive Approach for Longest Common Subsequence in Python*

The implementation of the pseudocode in chapter 2 for the recursive approach to solve LCS:

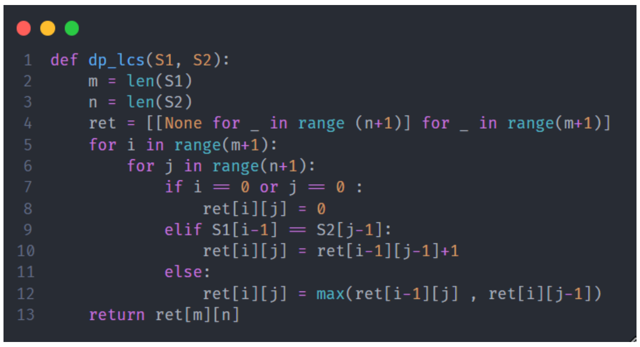


```python
def recursive_lcs(S1, S2, m, n):
    if m == 0 or n == 0:
        return 0
    elif S1[m-1] == S2[n-1]:
        return 1 + recursive_lcs(S1, S2, m-1, n-1)
    else:
        return max(recursive_lcs(S1, S2, m, n-1), recursive_lcs(S1, S2, m-1, n))

```

Figure 3. Recursive LCS Python Implementation

This implementation has complexity O(2n)

The implementation of the dynamic programming steps to solve LCS described in part A of this chapter in python:



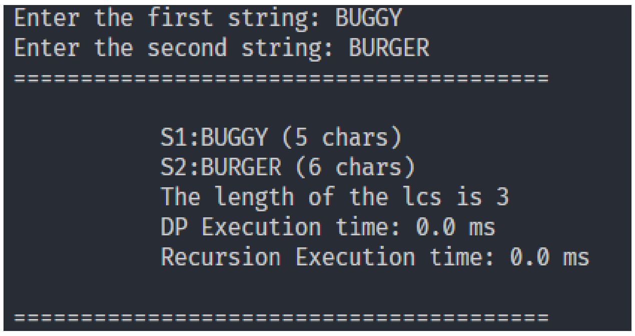
```python
def dp_lcs(S1, S2):
    m = len(S1)
    n = len(S2)
    ret = [[None for _ in range (n+1)] for _ in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                ret[i][j] = 0
            elif S1[i-1] == S2[j-1]:
                ret[i][j] = ret[i-1][j-1]+1
            else:
                ret[i][j] = max(ret[i-1][j] , ret[i][j-1])
    return ret[m][n]
```

Figure 4. dp_lcs Python Implementation (Source: Personal Documentation)

This implementation has O(m.n) complexity due to the looping of m and n. O(mn) is polynomial, meanwhile O(2n) is exponential.

Both functions are tested with different string length:

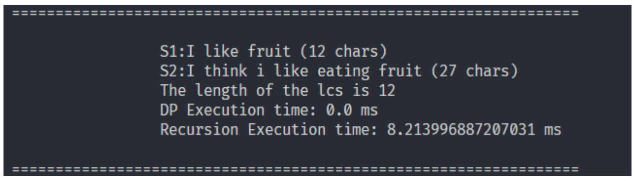
Case 1: "BUGGY" – "BUGGER"



```
Enter the first string: BUGGY
Enter the second string: BURGER
==========================================

          S1:BUGGY (5 chars)
          S2:BURGER (6 chars)
          The length of the lcs is 3
          DP Execution time: 0.0 ms
          Recursion Execution time: 0.0 ms

==========================================
```

Figure 5. Case 1 common subsequence (Source: Personal Documentation)

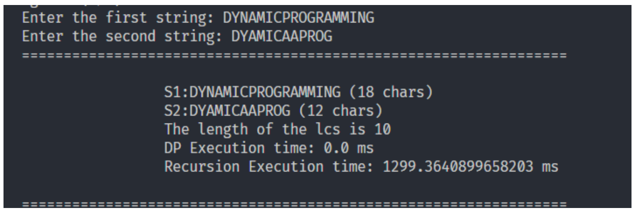
Case 2: “I like fruit” – “I think I like eating fruit”



```
==================================================================

                 S1:I like fruit (12 chars)
                 S2:I think i like eating fruit (27 chars)
                 The length of the lcs is 12
                 DP Execution time: 0.0 ms
                 Recursion Execution time: 8.213996887207031 ms

==================================================================
```

Figure 6. Case 2 common subsequence (Source: Personal Documentation)

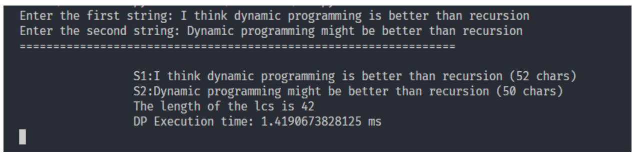
Case 3: “DYNAMICPROGRAMMING” – “DYAMICAAPROG”



```
Enter the first string: DYNAMICPROGRAMMING
Enter the second string: DYAMICAAPROG
================================================================

                S1:DYNAMICPROGRAMMING (18 chars)
                S2:DYAMICAAPROG (12 chars)
                The length of the lcs is 10
                DP Execution time: 0.0 ms
                Recursion Execution time: 1299.3640899658203 ms

================================================================
```

Figure 7. Case 3 common subsequence (Source: Personal Documentation)

Case 4: “I think dynamic programming is better than recusion” – “Dynamic programming might be better than recursion”



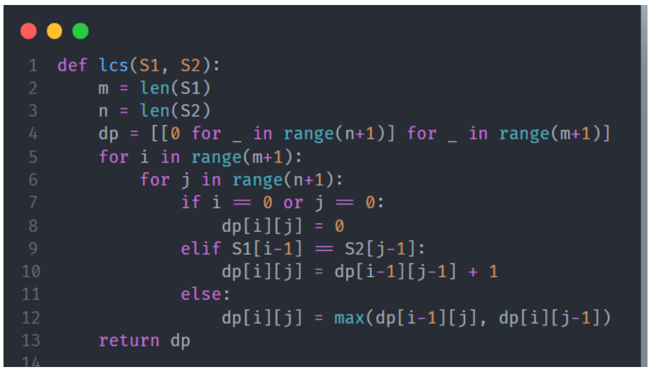
```
Enter the first string: I think dynamic programming is better than recursion
Enter the second string: Dynamic programming might be better than recursion
=================================================================

                 S1:I think dynamic programming is better than recursion (52 chars)
                 S2:Dynamic programming might be better than recursion (50 chars)
                 The length of the lcs is 42
                 DP Execution time: 1.4190673828125 ms
▌
```

Figure 8. Case 8 common subsequence (Source: Personal Documentation)

From the test cases above, it is evident that the hypotheses that the dynamic programming approach is a lot more efficient than the recursive approach. In case 1, where the string compared are short, recursive approach still performs well. However, as the strings get longer, the recursive approach performs badly giving a long execution time. With string length > 20, the recursive approach can’t finish its calculation in expected time. Meanwhile, the dynamic programming approach still performs well.

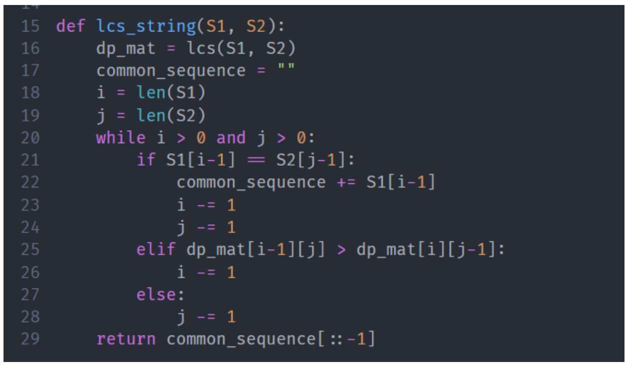
C. *Application of Dynamic Programming for Efficient Text-Based Files Comparison*

To show the difference between two text-based files, the python code implemented in the previous part is extended so that it returns the dp matrix using in the processing.



```python
def lcs(S1, S2):
    m = len(S1)
    n = len(S2)
    dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif S1[i-1] == S2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp
```

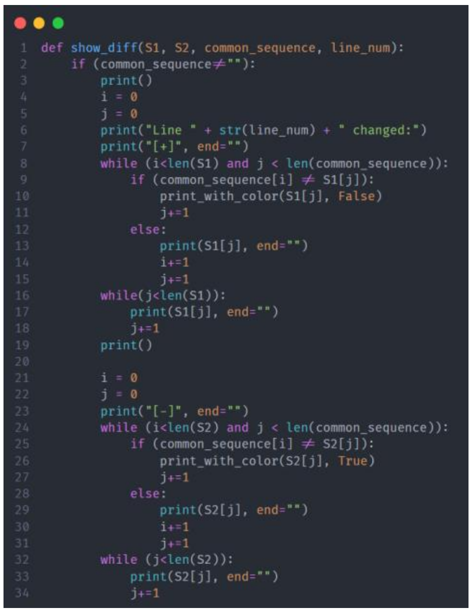
Figure 9. Modified dp lcs in Python (Source: Personal Documentation)



```python
def lcs_string(S1, S2):
    dp_mat = lcs(S1, S2)
    common_sequence = ""
    i = len(S1)
    j = len(S2)
    while i > 0 and j > 0:
        if S1[i-1] == S2[j-1]:
            common_sequence += S1[i-1]
            i -= 1
            j -= 1
        elif dp_mat[i-1][j] > dp_mat[i][j-1]:
            i -= 1
        else:
            j -= 1
    return common_sequence[::-1]
```

Figure 10. lcs_string Python Implementation (Source: Personal Documentation)

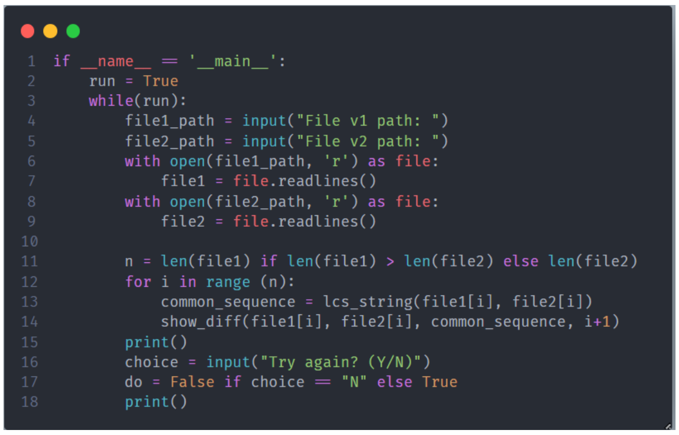
A new function lcs_string(S1, S2) is added, where the function first retrieves the dp matrix from the processing of S1 and S2 by lcs(S1, S2). Next, it searches the right most bottom most corner and store the characters one by one in the common_sequence string.



```python
def show_diff(S1, S2, common_sequence, line_num):
    if (common_sequence!=""):
        print()
        i = 0
        j = 0
        print("Line " + str(line_num) + " changed:")
        print("[+]", end="")
        while (i<len(S1) and j < len(common_sequence)):
            if (common_sequence[i] != S1[j]):
                print_with_color(S1[j], False)
                j+=1
            else:
                print(S1[j], end="")
                i+=1
                j+=1
        while(j<len(S1)):
            print(S1[j], end="")
            j+=1
        print()

        i = 0
        j = 0
        print("[-]", end="")
        while (i<len(S2) and j < len(common_sequence)):
            if (common_sequence[i] != S2[j]):
                print_with_color(S2[j], True)
                j+=1
            else:
                print(S2[j], end="")
                i+=1
                j+=1
        while (j<len(S2)):
            print(S2[j], end="")
            j+=1
```

Figure 11. show_diff implementation (Source: Personal Documentation) show_diff is also implemented to show the difference between to strings.

All of the functions above are used together in a main program that accepts two text-based files, and show the difference between the two by processing each line.



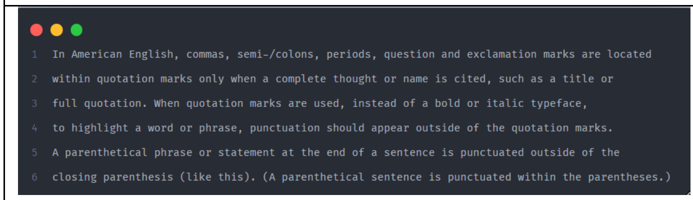
```python
if __name__ == '__main__':
    run = True
    while(run):
        file1_path = input("File v1 path: ")
        file2_path = input("File v2 path: ")
        with open(file1_path, 'r') as file:
            file1 = file.readlines()
        with open(file2_path, 'r') as file:
            file2 = file.readlines()

        n = len(file1) if len(file1) > len(file2) else len(file2)
        for i in range (n):
            common_sequence = lcs_string(file1[i], file2[i])
            show_diff(file1[i], file2[i], common_sequence, i+1)
        print()
        choice = input("Try again? (Y/N)")
        do = False if choice == "N" else True
        print()
```

Figure 12. Main Program Impelementation (Source: Personal Documentation)

The main program is used to compare two files to illustrate its implementation in version control.

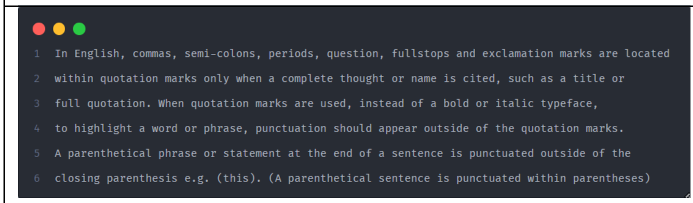
First test case (simple txt file):

file1-v1.txt



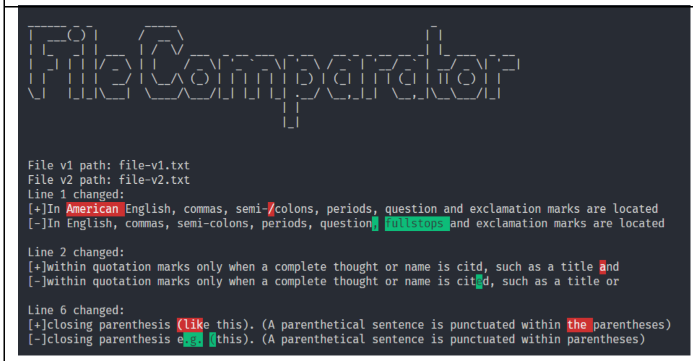
```
In American English, commas, semi-/colons, periods, question and exclamation marks are located
within quotation marks only when a complete thought or name is cited, such as a title or
full quotation. When quotation marks are used, instead of a bold or italic typeface,
to highlight a word or phrase, punctuation should appear outside of the quotation marks.
A parenthetical phrase or statement at the end of a sentence is punctuated outside of the
closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
```

File2-v2.txt



```
In English, commas, semi-colons, periods, question, fullstops and exclamation marks are located
within quotation marks only when a complete thought or name is cited, such as a title or
full quotation. When quotation marks are used, instead of a bold or italic typeface,
to highlight a word or phrase, punctuation should appear outside of the quotation marks.
A parenthetical phrase or statement at the end of a sentence is punctuated outside of the
closing parenthesis e.g. (this). (A parenthetical sentence is punctuated within parentheses)
```

Output



```
______ _ _        _____                                       _             
|  ___(_) |      /  __ \                                     | |            
| |_   _| | ___  | /  \/ ___  _ __ ___  _ __   __ _ _ __ __ _| |_ ___  _ __ 
|  _| | | |/ _ \ | |    / _ \| '_ ` _ \| '_ \ / _` | '__/ _` | __/ _ \| '__|
| |   | | |  __/ | \__/\ (_) | | | | | | |_) | (_| | | | (_| | || (_) | |   
\_|   |_|_|\___|  \____/\___/|_| |_| |_| .__/ \__,_|_|  \__,_|\__\___/|_|   
                                       | |                                  
                                       |_|                                  


File v1 path: file-v1.txt
File v2 path: file-v2.txt
Line 1 changed:
[+]In American English, commas, semi-/colons, periods, question and exclamation marks are located
[-]In English, commas, semi-colons, periods, question, fullstops and exclamation marks are located

Line 2 changed:
[+]within quotation marks only when a complete thought or name is citd, such as a title and
[-]within quotation marks only when a complete thought or name is cited, such as a title or

Line 6 changed:
[+]closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses)
[-]closing parenthesis e.g. (this). (A parenthetical sentence is punctuated within parentheses)
```

Second test case (python script):

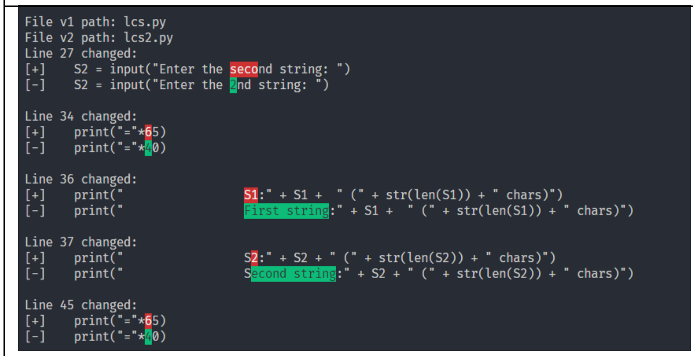
lcs.py

((accessible in https://github.com/chaerla/Text-Based-File-Comparator))

lcs2.py

((accessible in https://github.com/chaerla/Text-Based-File-Comparator))

Output



```
File v1 path: lcs.py
File v2 path: lcs2.py
Line 27 changed:
[+]    S2 = input("Enter the second string: ")
[-]    S2 = input("Enter the 2nd string: ")

Line 34 changed:
[+]    print("="*65)
[-]    print("="*40)

Line 36 changed:
[+]    print("                   S1:" + S1 +  " (" + str(len(S1)) + " chars)")
[-]    print("                   First string:" + S1 +  " (" + str(len(S1)) + " chars)")

Line 37 changed:
[+]    print("                   S2:" + S2 + " (" + str(len(S2)) + " chars)")
[-]    print("                   Second string:" + S2 + " (" + str(len(S2)) + " chars)")

Line 45 changed:
[+]    print("="*65)
[-]    print("="*40)
```

IV. Conclusion

Text-based file comparison is a critical task that can be effectively addressed using dynamic programming techniques. By formulating the comparison problem as the longest common substring (LCS) problem, we can leverage dynamic programming algorithms to achieve efficient and accurate results. Compared to the recursive approach, which has a complexity of O(n^2), the dynamic programming approach offers a significant improvement with a complexity of O(mn), making it suitable for comparing large texts.

The implementation of a file comparator using dynamic programming showcases its practicality and usefulness. It enables the detection of differences between files, paving the way for the development of various valuable tools. For instance, the comparator can be extended to create a plagiarism detector, helping to identify similarities between texts and detect instances of content reuse. Additionally, it can serve as a foundation for building a version control system, empowering developers to track changes, manage revisions, and collaborate effectively.

The efficiency and accuracy provided by dynamic programming in text-based file comparison open up possibilities for enhanced data analysis, content management, and decision-making. As technology advances, further advancements in file comparison algorithms and tools can be expected, ultimately leading to improved productivity and effectiveness across multiple domains.

Video Link at YouTube

https://youtu.be/YW8qu0iSze8

ACKNOWLEDGMENT

The author extends heartfelt gratitude to the following individuals:

1. The Almighty God, whose blessings and guidance have been instrumental in the successful completion of this paper. The author acknowledges the divine support that has been pivotal throughout the writing process.
2. The author's parents, whose unwavering support, encouragement, and belief in the author's abilities have been invaluable. Their constant presence and motivation have served as a driving force in the completion of this work.
3. Dr. Ir. Rinaldi Munir, M.T., Dr.Ir. Rila Mandala, and Dr. Nur Ulfa Maulidevi, esteemed professors of the IF2211 Algorithm Strategies course. The author expresses deep appreciation for their exceptional mentorship, insightful guidance, and the wealth of knowledge they shared throughout the academic journey. Their dedication and expertise have significantly enriched the author's understanding of the subject matter.

The author acknowledges and is grateful for the immeasurable contributions of these individuals, whose support and guidance have played a pivotal role in the successful completion of this paper.

REFERENCES

[1] Munir, Rinaldi. 2021. Program Dinamis.). [online] Available at: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf

[2] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), Introduction to Algorithms (2nd ed.), MIT Press & McGraw–Hill, ISBN 0-262-03293-7 . pp. 344.K. Elissa, “Title of paper if known,” unpublished.

[3] David Maier (1978). "The Complexity of Some Problems on Subsequences and Supersequences". J. ACM. ACM Press.

[4] Hirschberg, D. S. (1975). "A linear space algorithm for computing maximal common subsequences". Communications of the ACM.

DECLARATION

I hereby declare that this paper, which I have written, is my own original work and is not a summary, translation, or plagiarized version of someone else's paper.

Yogyakarta, 20 Mei 2023

[illegible]

Rachel Gabriela Chen