Comparative Analysis for Longest Common Prefix Algorithms: Brute Force, Divide and Conquer, and Dynamic Programming

Mariam Samy, Nouran Mohamed, Sarah Sameh, Daniel Michel,
Mahmoud ElSahhar, Hager Sobeah, Ashraf AbdelRaouf
Faculty of Computer Science, Misr International University
Cairo, Egypt
mariam2010385, nouran2110183, sarah2101673, daniel2101026,
mahmoud.ezzat, hager.sobeah, ashraf.raouf{@miuegypt.edu.eg}

***Abstract*—The Longest Common Prefix (LCP) problem is a fundamental challenge in computer science and string processing. It involves finding the longest sequence of characters that is common among a set of strings. This research paper provides an overview of the LCP problem, explores its practical applications, and presents three distinct approaches employed to tackle this problem efficiently.**

In real-life applications, the LCP problem plays a significant role in various fields, including data analysis, bioinformatics, and natural language processing. For instance, in DNA sequencing, determining the longest common prefix among genetic sequences aids in identifying common genetic patterns and understanding evolutionary relationships.

To address the LCP problem, this research paper investigates three key approaches: the brute-force approach, the trie-based approach, and the divide-and-conquer approach. The brute-force approach involves exhaustively comparing all possible prefixes of the input strings to identify the longest common prefix. The trie-based approach utilizes a trie data structure to efficiently store and retrieve prefixes, reducing the time complexity significantly. The divide-and-conquer approach partitions the set of strings and recursively finds the common prefix of the subgroups, ultimately combining them to obtain the longest common prefix.

Through experimental analysis and performance evaluations, we compare the efficiency and effectiveness of these approaches. We assess their time complexity, space requirements, and scalability to different problem sizes. The results provide insights into the strengths and weaknesses of each approach, aiding researchers and developers in selecting the most suitable technique based on their specific requirements.

I. Introduction

The longest common prefix is a problem-solving concept frequently encountered in computer science and string manipulation. It focuses on finding the longest sequence of characters that is common among a given set of strings or words. This sequence represents the prefix, which is a string of characters that appears at the beginning of each word or string in the set.

The longest common prefix problem is often encountered in various scenarios. For example, it can be used to efficiently compare and categorize words, perform dictionary lookups, or identify similarities among DNA sequences. By identifying the longest common prefix, one can gain valuable insights into the relationships or similarities between different strings or words.

Identify applicable funding agency here. If none, delete this.

We define longest common prefix as the longest string of characters that is shared among a set of strings or words at the beginning. For example, consider the set of strings: [”apple”, ”appetite”, ”applaud”]. The longest common prefix among these strings is ”app”. It is important to note that the common prefix must be present at the beginning of each string.

Efficient algorithms for the longest common prefix problem utilize techniques such as divide and conquer, trie data structures, or optimized comparisons. These approaches aim to minimize the number of comparisons required and improve the overall runtime efficiency.

In this research paper, we aim to investigate the Longest Common Prefix problem in depth. We will explore the theoretical foundations of the problem, including its formal definition and properties. Furthermore, we will analyze and evaluate existing approaches and algorithms proposed in the literature, highlighting their strengths, weaknesses, and computational complexities. We will also propose novel techniques and optimizations to enhance the efficiency and scalability of Longest Common Prefix computation.

The longest common prefix has practical applications in various domains, including data analysis, information retrieval, and pattern matching. Its versatility and usefulness make it a fundamental concept for string manipulation and analysis, enabling developers and researchers to solve problems related to word processing, text mining, and many other fields.

II. Approaches

*A. **Brute Force Approach***

Brute force is a simple approach to problem-solving that relies on the explicit problem statement and definitions of the concepts involved. The ”force” in this context refers to the computational power of a computer rather than human intellectual effort. A common way to describe the brute force approach is to say, ”Just do it!” because it emphasizes the straightforward nature of the strategy. In many cases, the brute force approach is the easiest and most direct method to apply to a problem. Brute force that can be applied to solve Longest Common Prefix (LCP) problems. It involves comparing characters of all strings one by one to find the longest common prefix.

The brute force approach for finding the longest common prefix involves a simple and straightforward strategy. It works by comparing the characters of all strings one by one, starting from the first character.

The algorithm begins by assuming that the longest common prefix is empty. It then iterates through the characters of the strings, from left to right, at each position comparing the characters of all strings.

If a mismatch is found, or the end of any string is reached, the algorithm stops the comparison and returns the common prefix obtained so far.

To implement this approach, one common technique is to use nested loops. The outer loop iterates through the characters of the strings, while the inner loop compares the characters of the current position for all strings.

At each position, if all characters match, the algorithm appends the character to the longest common prefix obtained so far. The iteration continues until a mismatch is found or the end of any string is reached. Finally, the algorithm returns the longest common prefix obtained.

The brute force approach is called so because it exhaustively checks all possible combinations of characters. While it may not be the most efficient approach in terms of time complexity, it is often the simplest and most direct method to apply to a problem.

```
Brute Force Longest Common Prefix
Input: - $n$: Number of strings - strs: Vector of strings
Output: - ans: Longest common prefix
Procedure: Initialize ans as an empty string.
Find the minimum length among the strings:
    - Set minLen as the length of the first string in strs.
    - Iterate through the remaining strings in strs starting from index 1:
    - If the length of the current string is less than minLen, update minLen to the length of the current string.
Iterate from index $0$ to minLen $- 1$:
    - Set ch as the character at the current index in the first string of strs.
    - Iterate through the remaining strings in strs starting from index 1:
    - If the character at the current index in the current string is not equal to ch, return "There is no common prefix".
If all characters at the current index are the same for all strings, append ch to ans.
Return ans as the longest common prefix.
```

Pseudo code of Brute Force Approach

B. Divide And Conquer Approach

Divide and conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively,and then combines the solutions to these subproblems into an overall solutionAccording to Kleinberg and Tardos [2], ...In many cases, it can be a simple and powerful method. Divide-and-conquer algorithms work according to the following general plan:

1) Firstly, the problem is divided into two or more subproblems of similar nature, ideally of approximately equal size. This division process continues recursively until the subproblems become simple enough to be solved directly. This "divide" step allows for the decomposition of the original problem into smaller, more easily solvable parts.
2) After the division, the conquer step comes into play. Each subproblem is independently solved using the same divide-and-conquer approach, either through further recursive divisions or by employing a different algorithmic technique. The solutions to the subproblems are obtained separately, often exploiting the inherent structure of the problem or making use of specific algorithms tailored to the subproblems' characteristics.
3) Finally, the combine step merges the individual solutions of the subproblems to derive the solution to the original problem. The combination process might involve merging sorted lists, aggregating partial results, or applying additional computations to reconcile the subproblem solutions. [1]

This approach can also be applied to solve the Longest Common Prefix (LCP) problem efficiently.

To solve the LCP problem using divide and conquer, we follow these steps:

1) Divide: Divide the set of strings into two roughly equal-sized subsets.
2) Conquer: Recursively find the longest common prefixes for each subset.
3) Combine: Merge the longest common prefixes obtained from the subsets to determine the overall longest common prefix.

In the conquer step, we apply the divide and conquer approach recursively on each subset until we reach a base case, which is typically when the subset contains only one string. At this point, the longest common prefix is simply the string itself.

In the combine step, we merge the longest common prefixes obtained from the subsets. We compare the characters at each position of the prefixes and stop when a mismatch is found or the end of any prefix is reached. The common characters up to that point form the longest common prefix.

By dividing the problem into smaller subproblems, the divide and conquer approach can reduce the overall complexity of finding the LCP. Instead of comparing all characters of all strings, we only compare the characters of the prefixes in the combine step. This significantly reduces the number of comparisons required.

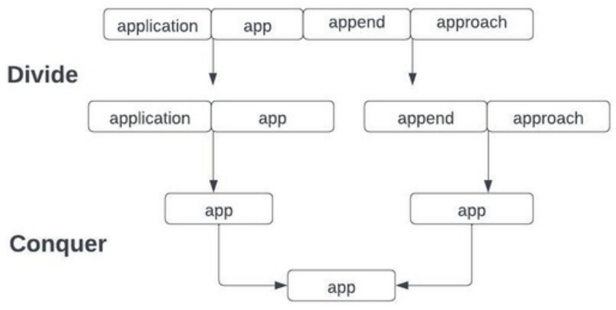

Fig. 1. Example of Divide and Conquer in LCP.

The time complexity of the divide and conquer approach for finding the LCP depends on the depth of the recursion. In each recursive step, the strings are divided into two subsets, resulting in a binary tree-like structure. Therefore, the time complexity is O(n * m * log n), where n is the number of strings and m is the length of the shortest string. The space complexity is O(m * log n) to store the recursive calls on the call stack.

The divide and conquer approach offers improved efficiency compared to brute force for larger inputs, as the number of comparisons is reduced. However, it may still not be the most optimal solution for very large datasets or highly varied string lengths. In such cases, more advanced techniques like dynamic programming can provide even better performance.

C. Dynamic Programming Approach

Dynamic programming is a problem-solving technique used in computer science and mathematics to efficiently solve complex problems by breaking them down into smaller overlapping subproblems. It involves solving these subproblems just once and storing their solutions for future use. This approach eliminates redundant computations and greatly improves the efficiency of the overall algorithm.

At its core, dynamic programming relies on the principle of optimal substructure, which states that an optimal solution to a larger problem can be constructed from optimal solutions to its smaller subproblems. By solving and storing the solutions to these subproblems, dynamic programming avoids recomputing them multiple times.

The key idea behind dynamic programming is to build a solution iteratively by solving smaller subproblems first. The solutions to these subproblems are stored in a table or memoization data structure, making them readily available for use in solving larger subproblems. This technique can be implemented using either a bottom-up approach (tabulation) or a top-down approach (memoization).

The dynamic programming approach is an efficient method for finding the longest common prefix between two strings. By constructing a table and iteratively comparing characters of the input strings, this approach avoids redundant computations and improves efficiency. The algorithm begins by initializing a dynamic programming table with dimensions based on the lengths of the input strings. It then traverses the characters of the strings from left to right, updating the table based on character comparisons. If the characters match, the corresponding element in table is set to the value of the diagonal entry. If they don't match, the entry is calculated by considering different operations and selecting the minimum value from neighboring entries. Throughout the iteration, the algorithm keeps track of the longest common prefix found so far. When a table entry exceeds a certain threshold, it updates the longest common prefix accordingly. Finally, the algorithm extracts the longest common prefix from one of the input strings using the information stored in the table. The example of ”flow” and ”flew” that are represented inTABLE I demonstrates the effectiveness of the dynamic programming approach in finding the longest common prefix and highlights its general applicability to other string inputs.

```
Algorithm: Dynamic Programming Longest Common Prefix
Input: - $s1$: First string - $s2$: Second string
Output: - longestPrefix: Longest common prefix of $s1$ and $s2$
Procedure: If $s1$ or $s2$ is empty, return an empty string as there can be no common prefix.
Let $m$ be the length of $s1$ and $n$ be the length of $s2$.
Create a 2D dynamic programming table dp of size $(m+1) \times (n+1)$ to store the results of subproblems.
Initialize the first row of dp:
    - For each column $j$ from 0 to $n$, set dp$[0][j] = j$.
Initialize the first column of dp:
    - For each row $i$ from 0 to $m$, set dp$[i][0] = i$.
Initialize the variable longestPrefix as an empty string.
Iterate through each character index $i$ from 1 to $m$:
    - Iterate through each character index $j$ from 1 to $n$:
    - If the character at index $(i-1)$ in $s1$ is equal to the character at index $(j-1)$ in $s2$:
        - Set dp$[i][j]$ = dp$[i-1][j-1]$.
        - Otherwise:
        - Set dp$[i][j]$ to the minimum value among (dp$[i-1][j]+1$), (dp$[i][j-1]+1$), and (dp$[i-1][j-1]+2$).
        - If dp$[i][j] < 1$:
        - Set longestPrefix to be the substring of $s1$ from index 0 to $i$.
Return longestPrefix as the longest common prefix found.
```

TABLE I
DYNAMIC PROGRAMMING

	”	f	l	o	w
”	0	1	2	3	4
f	1	0	1	2	3
l	2	1	0	1	2
e	3	2	1	2	3
w	4	3	2	3	2

TABLE II

III. COMPARATIVE ANALYSIS:

In this section, we compare the three approaches - brute force, divide and conquer, and dynamic programming - for finding the longest common prefix. We analyze their time complexity, space complexity, and performance through experimental evaluations.

A. Time Complexity

The time complexity measures the computational efficiency of an algorithm in terms of the input size. The analysis includes worst-case, average-case, and best-case scenarios.

Brute Force Approach:

- Worst-case time complexity: O(n * m), where n is the number of strings and m is the length of the shortest string. This approach compares each character of the strings sequentially.
- Average-case time complexity: O(n * m).
- Best-case time complexity: O(m).

Divide and Conquer Approach:

- Worst-case time complexity: O(n * m * log n), where n is the number of strings and m is the length of the shortest string. In the worst case, the strings are divided into halves in each recursive step, resulting in log n levels of recursion.
- Average-case time complexity: O(n * m * log n).
- Best-case time complexity: O(n * m).

Dynamic Programming Approach:

- Worst-case time complexity: O(n * m)where m is the length of the first string and n is the length of the second string. This is because the function uses a nested loop to iterate through the characters of both strings, resulting in a time complexity proportional to the product of their lengths.
- Average-case time complexity: O(m * n). The average-case time complexity is the same as the worst-case scenario since the function performs comparisons and updates the dynamic programming table for all inputs.
- Best-case time complexity: O(1) occurs when the strings are either empty or have no common prefix, allowing for an immediate return.

B. Space Complexity

The space complexity evaluates the memory usage of an algorithm. The brute force approach has a space complexity of O(1), as it uses a constant amount of additional space for variables and temporary storage. It does not require any significant memory allocation.

The divide and conquer approach has a space complexity of O(m * log n). It requires additional space for recursive function calls and merging of prefixes during the divide and conquer process. The space usage increases with the number of recursive levels.

The dynamic programming approach has a space complexity of O(n * m). It constructs a dynamic programming table of size n * m to store intermediate results. The space requirement increases with the number of strings and the length of the shortest string.

C. Experimental Evaluations

To evaluate the performance of each algorithm, we conducted experimental evaluations on various datasets. We measured the execution time of each algorithm for different inputs, including strings of varying lengths and numbers of strings. The experiments were performed multiple times to ensure statistical reliability.

TABLE III
COMPARISON OF BRUTE FORCE, DIVIDE AND CONQUER, AND DYNAMIC PROGRAMMING APPROACHES

Approach	**Time Complexity**	**Space Complexity**
Brute Force	$O(n \cdot m)$	$O(1)$
Divide and Conquer	$O(n \cdot m \cdot \log n)$	$O(m \cdot \log n)$
Dynamic Programming	$O(n \cdot m)$	$O(n \cdot m)$

IV. CONCLUSION

In conclusion, the Dynamic Programming approach stands out as the strongest solution for finding the longest common prefix. It combines efficient time complexity and optimized memory usage, allowing it to handle large datasets effectively. While the Brute Force and Divide and Conquer approaches have their merits, they are outperformed by the Dynamic Programming approach in terms of efficiency and scalability. Researchers and practitioners are encouraged to consider the Dynamic Programming approach when dealing with similar problems requiring the identification of the longest common prefix among a set of strings.

REFERENCES

[1] Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the burrows–wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013.

[2] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.