Shahjalal University of Science and Technology
Department of Computer Science and Engineering

A dynamic method for addition in all pair suffix-prefix matching problem

ALFEH SANI
Reg. No.: 2018331119
4^{th} year, 2^{nd} Semester

MAHBUBUL HASAN
Reg. No.: 2018331123
4^{th} year, 2^{nd} Semester

Department of Computer Science and Engineering

Supervisor

DR. HUSNE ARA CHOWDHURY
Associate Professor
Department of Computer Science and Engineering

26th February, 2024

A dynamic method for addition in all pair suffix-prefix matching problem

A Thesis submitted to the Department of Computer Science and Engineering, Shahjalal University of Science and Technology, in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering.

By

Alfeh Sani
Reg. No.: 2018331119
4^{th} year, 2^{nd} Semester

Mahbubul Hasan
Reg. No.: 2018331123
4^{th} year, 2^{nd} Semester

Department of Computer Science and Engineering

Supervisor

Dr. Husne Ara Chowdhury
Associate Professor
Department of Computer Science and Engineering

26th February, 2024

Recommendation Letter from Thesis Supervisor

The thesis entitled *A dynamic method for addition in all pair suffix-prefix matching problem* submitted by the students

1. Mahbubul Hasan
2. Alfeh Sani

is under my supervision. I, hereby, agree that the thesis can be submitted for examination.

Signature of the Supervisor:
Name of the Supervisor: Dr Husne Ara Chowdhury
Date: 26th February, 2024

Certificate of Acceptance of the Thesis

The thesis entitled *A dynamic method for addition in all pair suffix-prefix matching problem* submitted by the students

1. Mahbubul Hasan
2. Alfeh Sani

on 26th February, 2024, hereby, accepted as the partial fulfilment of the requirements for the award of their Bachelor's Degrees.

Head of the Dept.	Chairman, Exam. Committee	Supervisor
MD Masum	MD Masum	Dr. Husne Ara Chowdhury
Professor	Professor	Assistant Professor
Department of Computer Science and Engineering	Department of Computer Science and Engineering	Department of Computer Science and Engineering

Abstract

String matching is a fundamental problem in computer science with numerous applications in areas such as information retrieval, bio-informatics, and natural language processing. One classical variant of this problem is the "All-Pair Suffix-Prefix Matching[1]," which involves finding the longest matching prefixes of some strings with the suffixes of other strings, across all possible pairs. The existing solutions for this problem are limited to offline scenarios, where the entire data set is known beforehand. But whenever new strings are inserted, the model of existing solutions needs to be run again which is computationally slow. In our proposed model, we introduce a novel approach to address this limitation by proposing a dynamic model for online string insertion. Our proposed model not only efficiently computes all-pair suffix-prefix matches with existing strings but also seamlessly accommodates new string insertions.

Our model[2] uses advanced data structures and algorithms to enable real-time matching when new strings are inserted. The key challenge lies here is to handle dynamic updates while optimizing time complexity. In conclusion, this thesis presents a significant advancement in the field of online string matching by introducing a dynamic all-pair suffix-prefix matching[1] model.

Keywords: Suffix Array[3], Treap(cartesian tree)[4], APSP[1], RMQ[5], Segment Tree [6]

Acknowledgements

We would like to start by thanking our supervisor Dr. Husne Ara Chowdhury, Associate Professor, Department of CSE, SUST for her admirable support and continuous motivation.

We express our deepest gratitude to our co-supervisor Md Saiful Islam, Assistant Professor, Department of CSE, SUST for his effort, advice, and the support he provided at the time of our need. Their support encouraged us to make progress in the correct direction. Their constant guidance enabled us to conduct our research successfully.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

All pair suffix prefix matching problem refers to finding the longest matching prefixes of some strings with the suffixes of other strings, across all possible pairs of strings from a set of strings. In more detail, given a collection of n strings $s_1, s_2, s_3, \ldots, s_n$, the APSP is the problem of finding for all pairs s_i and s_j, the longest suffix of s_i which is also prefix of s_j.

But the problem arises when we want to find a matching for a new string $s^{'}$, which is not present in the set, we have to rebuild the whole model and need to compare it with every string that was present in the set. Thus the time complexity increases by O(n) for every new insertion, where n is the size of the set before insertion.

Some solutions which are non-optimal have been proposed earlier in the past few years. A significant reduction of memory consumption was achieved by Rachid et al in the APSP matching problem, which presented new space-efficient algorithms using compressed data structures. [11]

In our thesis, we propose an optimal algorithm that is faster and space efficient (when insertion of a new string) in practice using suffix arrays. Experiments have also shown that our proposed algorithm may be a good practical solution when searching for suffix−prefix overlaps of small length. Also, our proposed algorithm works faster when we try to insert a new string into the set of strings and try to find prefix-suffix matching which is good compared to the older algorithms.

1.1 Motivation

Algorithmic Research and Optimization: Insertion of a new string in an APSP matching problem, involves designing efficient algorithms and data structures to find all matching suffix-prefix pairs across large sets of strings. Exploring novel algorithmic approaches and optimization techniques to solve this problem could contribute to the advancement of string algorithms and data processing.

Bioinformatics and Sequence Analysis: In bioinformatics, DNA and protein sequences are analyzed to understand genetic information. The suffix-prefix matching problem could have applications in identifying common patterns or subsequences between different sequences, aiding in understanding genetic relationships and evolutionary patterns.

Data Compression and Pattern Recognition: Efficiently identifying matching patterns in strings is crucial in data compression techniques like Burrows-Wheeler Transform (BWT) and its variations. Researching the suffix-prefix matching problem could lead to improved compression algorithms and better data representation.

Pattern Matching in Image Processing: In some cases, strings can be thought of as one-dimensional representations of data. Exploring the extension of the suffix-prefix matching problem to multidimensional data could have applications in image or signal processing.

1.2 Objective

The primary objective of our thesis is to develop an efficient and optimized algorithm for the All Pair Suffix Prefix Matching (APSP) problem that significantly reduces the time required for computations when new strings are inserted into the existing set. The main focus lies in addressing the challenges associated with rebuilding the model and recalculating the matches after each insertion.

To achieve this goal the following objective needs to be pursued:

1. Develop a method that speeds up the process of rebuilding the APSP model when new strings are inserted and finding matches when new strings are added. The aim is to minimize the extra time required for each new insertion.

2. Implement suffix arrays in a practical way to enhance match calculations and allow quick updates when new strings are introduced.

3. Check how well the proposed algorithm works as the set of strings grows which we going to implement in the upcoming semester.

1.3 Challenges

Until now, numerous algorithms have been proposed to address the All-Pairs Shortest Path (APSP) problem. Different researchers have employed various data structures and algorithms, such as Suffix Array, Suffix Tree, Aho-Corasick, etc., in their attempts to solve this problem. Over time, researchers have iteratively improved upon existing algorithms, adapting them to different computational environments. All existing solutions operate in an offline manner, and there is currently no algorithm designed for online string processing in the context of APSP.

Motivated by this gap, we aimed to develop an algorithm capable of efficiently solving the APSP problem as strings arrive online. Throughout this process, we explored various approaches, learning about diverse data structures and algorithms, including Set, Vector, Segment Tree, Range Minimum Query (RMQ), Treap, Binary Search Tree (BST), and Online Suffix Array, among others. One of our initial ideas involved a heuristic solution using a Segment Tree, tailored for randomized strings. However, due to its challenging implementation and high constant factor, we abandoned this approach.

Subsequently, we shifted our focus to the current idea, centred around Online Suffix Array. Dividing our task into two parts, we successfully tackled the first part but encountered challenges with the second. Despite contemplating various approaches, none proved to be effective. It was at this juncture that we considered the possibility of reversing the strings and constructing another Online Suffix Array to address the problem as the first task. Implementing this idea posed coding challenges, requiring substantial modifications. After persistent debugging efforts, we resolved all code errors, successfully completing both tasks.

Chapter 2

Related works

The problem of finding all pairs of suffix-prefix matches is well-studied in stringology. This problem was first solved in 1992 by Gusfield et al.[12] They present an algorithm using suffix tree that solves the problem in $O(m + k^2)$ time, for any fixed alphabet. Here, the alphabet size is $O(m)$ and the number of strings is k. Although suffix trees play a prominent role in algorithmics, but they suffer from two drawbacks:

1. Although being asymptotically linear, the space consumption of a suffix tree is substantial. Even with improved implementations, it still requires approximately 20 bytes per input character in the worst case.

2. In most applications, the suffix tree encounters poor memory reference locality, leading to a significant loss of efficiency on cached processor architectures.

In 2010, Enno Ohlebusch and Simon Gog proposed a solution that addresses the aforementioned problems through the use of a generalized enhanced suffix array.[13] This approach requires only 8 bytes per input character. Experimental results demonstrate that it utilizes approximately three times less space and is about three times faster than the algorithm employing a generalized suffix tree. Another advantage is that this algorithm is simpler to implement than the previous one.

In 2014, Maan Haj Rachid, Qutaibah Malluhi, and Mohamed Abouelhoda published a paper that further optimizes the solution to the all-pair suffix-prefix matching problem using a suffix tree.[14] The drawback of the initial paper is its consumption of 20 bytes of memory for each input character. Consequently, this algorithm proves unsuitable when handling substantial volumes of

data.

The mentioned paper specifically targets this concern and enhances the memory complexity of the suffix tree. Additionally, the authors propose a parallel algorithm designed to operate efficiently in a multithreaded environment.

Practical solutions, while not optimal, have emerged in recent years. Rachid et al. achieved a notable reduction in memory consumption by introducing space-efficient algorithms that utilize compressed data structures. However, their experimental results revealed that their solutions are roughly 100 times slower in practice compared to previous methods.

More recently, Rachid and Malluhi tackled the problem more efficiently using a compact prefix tree. In 2016, William H.A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza proposed a solution that addresses the problem locally for each string.[15] Their approach involves scanning the enhanced suffix array in reverse to avoid processing suffixes that do not qualify as candidates for suffix-prefix matching. Empirical evaluations demonstrated that their algorithm is over two times faster and more space-efficient than the method proposed by Ohlebusch and Gog.

Furthermore, other papers have explored solutions that build upon alternative data structures. In 2022, Grigorios Loukides and Solon P. Pissis presented a solution based on Aho-Corasick, which also effectively addresses the problem[16].

All the algorithms we have mentioned operate in an offline manner. Consider a scenario where we possess an extensive collection of strings and have determined all the pairs with maximum matching suffix-prefix. Now, imagine that a new string is introduced. If our goal is to identify the maximum matching suffix-prefix between this new string and all the previously existing strings, we would be required to re-run the entire model. However, this would lead to significantly higher complexity, which is the challenge posed by the existing model.

In the next chapter, we'll investigate an existing model based on the suffix array and examine how its complexity increases significantly.

Chapter 3

Background Study

We have to learn some algorithms and data structures[6] for this paper. The algorithms that we are learning include Segment Tree, Suffix Array, LCP array, Sparse Table, Queue, Array, and Treap. Detailed information about these algorithms is provided below.

3.1 Segment Tree [7]

A Segment Tree is a data structure that stores information about array intervals as a tree. This allows answering range queries over an array efficiently, while still being flexible enough to allow quick modification of the array. Key characteristics and concepts of a Segment Tree:

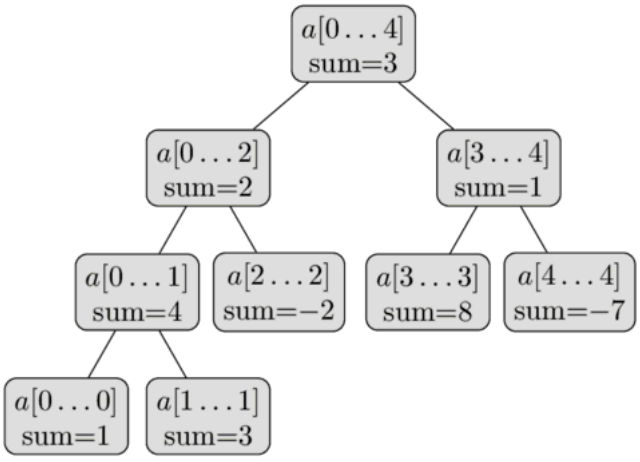

Figure 3.1: Segment Tree

3.1.1 Hierarchical Structure:

It's a type of binary tree where each leaf node represents an individual element from the input array. Each non-leaf node represents a segment (a range) of the array, combining the values of its child segments.

3.1.2 Construction:

The process of building a Segment Tree involves recursive partitioning of the input array. At each step, the array is divided into two halves and this process continues until each leaf node represents a single element.

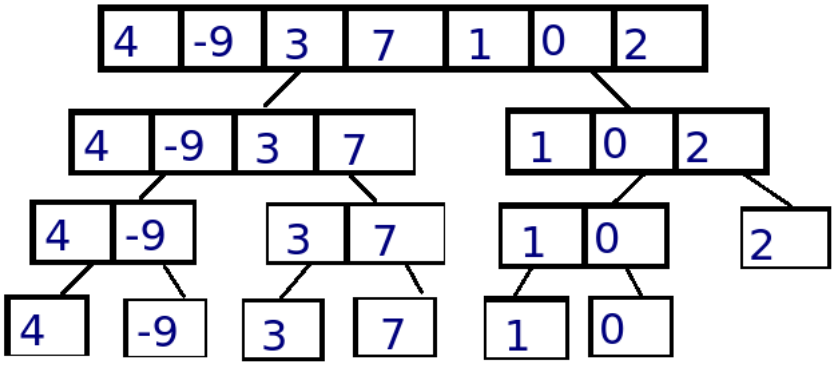

Figure 3.2: Segment Tree construction

3.1.3 Range Queries

To perform a range query (e.g., find the minimum/maximum/sum) over a specific range [L, R] in the original array, the Segment Tree efficiently combines information from various nodes in the tree that correspond to segments intersecting with [L, R]. This is achieved by traversing the tree from the root to the leaf nodes, considering only the relevant segments.

3.1.4 Updates

If the original array is subject to updates (element value changes), the Segment Tree can be updated efficiently to reflect these changes. When an update is performed, the corresponding leaf node is updated, and the change propagates upward through the tree, updating affected nodes.

Algorithm 1 Basic Segment Tree Pseudocode

```
function NodeConstructor
    return {sum = 0}                          ▷ Initialize a new node with a sum value of 0.
end function
function SegmentTreeConstructor(_n)
    t ← array of nodes with size 4 · _n + 1
    n ← _n
    return {t = t, n = n}          ▷ Initialize a new segment tree with an array of nodes and size information.
end function
function Merge(a, b)
    c ← NodeConstructor
    c.sum ← a.sum + b.sum
    return c          ▷ Combine two nodes by creating a new node with the sum of their values.
end function
function Query(n, l, r, le, ri)
    c ← NodeConstructor
    if le > r or le > ri or l > ri then
        return c                              ▷ If the range is invalid, return a default node.
    end if
    if l ≥ le and r ≤ ri then
        return t[n]   ▷ If the current segment is fully within the query range, return the stored node.
    end if
    mi ← (l + r)/2
    a ← Query(2 · n, l, mi, le, ri)
    b ← Query(2 · n + 1, mi + 1, r, le, ri)
    return Merge(a, b)                        ▷ Merge the results from left and right child nodes.
end function
function Update(n, l, r, ind, vala)
    if ind < l or ind > r then
        return                        ▷ If the index is outside the current segment, do nothing.
    end if
    if l == r then
        t[n].sum ← t[n].sum + vala                ▷ Update the leaf node with the new value.
        return
    end if
    mi ← (l + r)/2
    Update(2 · n, l, mi, ind, vala)
    Update(2 · n + 1, mi + 1, r, ind, vala)
    t[n] ← Merge(t[2 · n], t[2 · n + 1])    ▷ Update the current node by merging the updated children.
    return
end function
```

3.1.5 Time Complexity

Both range queries and updates in a Segment Tree have time complexities of $O(\log N)$, where N is the number of elements in the original array. This makes Segment Trees an attractive choice for problems requiring efficient querying and updating.

Segment Trees are widely used in competitive programming and algorithmic problem-solving due to their versatility in solving a wide range of problems, including finding minimum/maximum values, sum queries, and more complex queries.

3.2 Sparse Table: [5]

To perform range queries such as **Range Minimum Query** [17] Sparse Table is an efficient data structure which can answer queries in O(1). The only drawback of this data structure is, that it can only be used on immutable arrays. This means, that the array cannot be changed between two queries. If any element in the array changes, the complete data structure has to be recomputed. The main idea behind Sparse Tables is to precompute all answers for range queries with a power

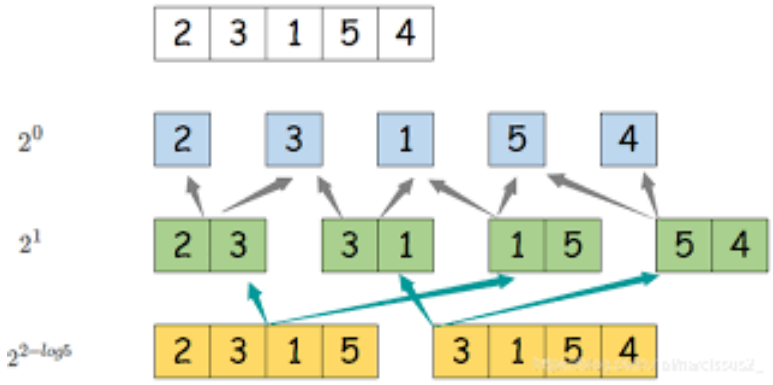

Figure 3.3: Sparse Table Example

of two lengths. Afterwards, a different range query can be answered by splitting the range into ranges with the power of two lengths, looking up the precomputed answers, and combining them to receive a complete answer.

Algorithm 2 Sparse Table Construction

```
1: function SparseTable(A[1...n])
2:     N ← size of A
3:     K ← ⌈log2(N)⌉
4:     table[1...N][0...K]                          ▷ Initialize a 2D array for sparse table
5:     for i ← 1 to N do
6:         table[i][0] ← A[i]                       ▷ Initialize the first column with the array values
7:     end for
8:     for j ← 1 to K do
9:         for i ← 1 to N do
10:            if i + 2^j − 1 ≤ N then
11:                table[i][j] ← min(table[i][j − 1], table[i + 2^(j−1)][j − 1])   ▷ Compute the minimum value for each range
12:            end if
13:        end for
14:    end for
15:    return table
16: end function
```

Algorithm 3 Sparse Table Query

```
1: function query(table, A, L, R)
2:     k ← ⌊log2(R − L + 1)⌋                        ▷ Find the maximum power of 2 that fits the range
3:     return min(table[L][k], table[R − 2^k + 1][k])   ▷ Return the minimum of the two ranges
4: end function
```

3.3 Suffix Array [3]

A suffix array is a data structure used in string processing and pattern matching that efficiently encodes the lexicographic order of all suffixes of a given string. It provides a powerful tool for solving a variety of string-related problems, including substring searching, string comparisons, and more. The main idea behind a suffix array is to create an array that represents the starting positions of all suffixes of a string, ordered in lexicographic order.

3.3.1 Example: Suffix Array

Let's consider a simple string S = "banana". We'll construct the suffix array for this string to demonstrate how it works.

Original string: "banana"

Suffixes:

- "banana" (Starting index: 1)
- "anana" (Starting index: 2)
- "nana" (Starting index: 3)
- "ana" (Starting index: 4)
- "na" (Starting index: 5)
- "a" (Starting index: 6)

After sorting them, their order will be:

- "a" (Starting index: 6)
- "ana" (Starting index: 4)
- "anana" (Starting index: 2)
- "banana" (Starting index: 1)
- "na" (Starting index: 5)
- "nana" (Starting index: 3)

That's how the suffix array works. It sorts all the suffixes according to lexicographical order.

3.3.2 Construction[8]

Given a string of length n, the suffix array is constructed by sorting all the suffixes of the string. Each element of the suffix array represents the starting index of a suffix in the original string. The sorting process is typically done using comparison-based sorting algorithms like quicksort, and mergesort, or specialized algorithms like the DC3 algorithm.

3.3.3 Advantages

The key advantage of a suffix array is that it allows for efficient substring search and comparison operations. Since the array is sorted lexicographically, finding a substring or comparing two substrings can be achieved in logarithmic time using binary search techniques.

3.3.4 Applications

- **Substring Search**: Given a query string, you can efficiently locate occurrences of the query in the original text using binary search on the suffix array. This allows for substring searches in logarithmic time.
- **Longest Common Prefix (LCP) Array**: The LCP array is another array associated with the suffix array that stores the length of the longest common prefix between consecutive suffixes. The LCP array has various applications in string algorithms, such as in finding repeated substrings.
- **Pattern Matching**: Suffix arrays can be used for pattern matching and text indexing. They provide a foundation for more advanced data structures like the suffix tree and Burrows-Wheeler Transform.

3.3.5 Space Complexity

The space complexity of a suffix array is usually proportional to the length of the original string. However, more space-efficient data structures, like enhanced suffix arrays (ESA), can reduce memory requirements while still providing similar functionality.

Algorithm 4 Sort Cyclic Shifts

```
1: function SortCyclicShifts(s: string, n: int, alphabet: int = 256) → vector of integers
2:     p, c ← vectors of length n, initialized with zeros
3:     cnt ← vector of length max(alphabet, n), initialized with zeros
4:     for i ← 0 to n − 1 do                          ▷ Count occurrences of each character
5:         cnt[s[i]] ← cnt[s[i]] + 1
6:     end for
7:     for i ← 1 to alphabet − 1 do                   ▷ Calculate cumulative count
8:         cnt[i] ← cnt[i] + cnt[i − 1]
9:     end for
10:    for i ← 0 to n − 1 do                          ▷ Generate initial order using counting sort
11:        p[cnt[s[i]] − 1] ← i
12:        cnt[s[i]] ← cnt[s[i]] − 1
13:    end for
14:    c[p[0]] ← 0                                    ▷ Assign classes to suffixes
15:    classes ← 1
16:    for i ← 1 to n − 1 do
17:        if s[p[i]] ≠ s[p[i − 1]] then
18:            classes ← classes + 1
19:        end if
20:        c[p[i]] ← classes − 1
21:    end for
22:    pn, cn ← vectors of length n
23:    for h ← 0 to ⌊log2 n⌋ do                       ▷ Iterate in powers of 2
24:        for i ← 0 to n − 1 do                      ▷ Generate new order using the previous order
25:            pn[i] ← (p[i] − 2^h + n) mod n
26:        end for
27:        cnt ← vector of length classes, initialized with zeros
28:        for i ← 0 to n − 1 do                      ▷ Reset and update cumulative count
29:            cnt[c[pn[i]]] ← cnt[c[pn[i]]] + 1
30:        end for
31:        for i ← 1 to classes − 1 do
32:            cnt[i] ← cnt[i] + cnt[i − 1]
33:        end for
34:        for i ← n − 1 to 0 do                      ▷ Generate new order of suffixes
35:            p[cnt[c[pn[i]]] − 1] ← pn[i]
36:            cnt[c[pn[i]]] ← cnt[c[pn[i]]] − 1
37:        end for
38:        cn[p[0]] ← 0                               ▷ Assign new classes to suffixes
39:        classes ← 1
40:        for i ← 1 to n − 1 do
41:            cur ← {c[p[i]], c[(p[i] + 2^h) mod n]}
42:            prev ← {c[p[i − 1]], c[(p[i − 1] + 2^h) mod n]}
43:            if cur ≠ prev then
44:                classes ← classes + 1
45:            end if
46:            cn[p[i]] ← classes − 1
47:        end for
48:        c ↔ cn
49:    end for
50:    return p
51: end function
```

Algorithm 5 Suffix Array Construction

```
1: function SuffixArrayConstruction(s: string) → vector of integers
2:     s ← s + "$"                                   ▷ Add a sentinel character
3:     n ← length of s
4:     sorted_shifts ← SortCyclicShifts(s, n)
5:     sorted_shifts.erase(sorted_shifts.begin())    ▷ Remove index of sentinel character
6:     return sorted_shifts
7: end function
```

3.3.6 Limitations

While suffix arrays are efficient for certain tasks, they may not be the best choice for all situations. Constructing a suffix array may require additional memory, and building the array can be relatively complex and time-consuming. Additionally, the suffix array doesn't handle dynamic text insertion or deletion efficiently, which is a limitation in applications involving frequently changing text.

In summary, a suffix array is a versatile data structure that plays a crucial role in solving various string-related problems, offering efficient substring search, comparison, and other useful operations. It forms the foundation for more advanced structures and algorithms designed to handle large amounts of text efficiently.

3.4 Binary Search Tree [9]

A Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node. This hierarchical structure allows for efficient searching, insertion, and deletion operations on the data stored in the tree. Binary Search Tree can be used for handling various kinds of complex operations. We can do operations like Insert, Delete, Update and Query in logarithmic time complexity. It can be used in heavy data structures like Treap, Online Suffix Array and various STLs[18]. [9]

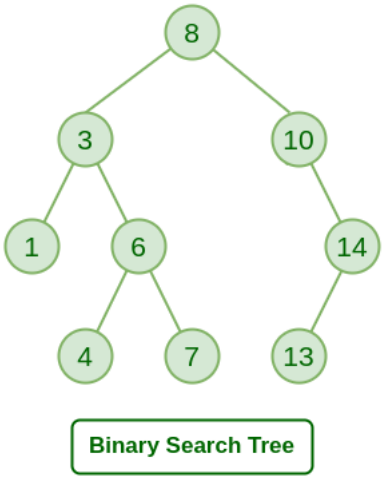

Figure 3.4: Binary Search Tree [9]

3.5 Onlilne Suffix Array [10]

The term "Online" implies that the string to which the suffix array corresponds can change over time. In other words, you can add or remove characters from the string, and the suffix array should be able to adapt to these changes efficiently.

Within the online suffix array, a character is either added or removed at the front of the existing string. This operation leaves all existing suffixes unchanged, only introducing a new suffix into the suffix array. Similar to the traditional suffix array, the online variant also maintains arrays for the Longest Common Prefix (LCP), Suffix Array (SA), Inverse Suffix Array (ISA), and includes a tag value for each suffix. Leveraging the existing values for the LCP, SA, ISA, and tags of the existing suffixes, the corresponding values for the newly inserted suffix can be efficiently calculated. To expedite these calculations, the online suffix array employs a specialized Binary Search Tree (BST), functioning akin to a treap. The use of a BST allows for the logarithmic time complexity in calculating all these values.

An Online suffix array maintains the sorted order of suffixes as the string it represents is modified. Achieving this efficiently is crucial for applications like text indexing, bioinformatics, and data compression, where frequent updates to the underlying string occur.

3.6 The Longest Common Prefix (LCP) Table [8]

The Longest Common Prefix (LCP) table is an important data structure constructed along with the Suffix Array during string processing tasks. The LCP table stores information about the lengths of the longest common prefixes between consecutive sorted suffixes of a given string. Let $Sa[i]$ be the i-th sorted suffix, and $Sa[i+1]$ be the $(i+1)$-th sorted suffix. Then $LCP[i]$ should be equal to the maximum matched prefix between two suffixes, $Sa[i]$ and $Sa[i+1]$. It provides valuable insights into the similarities between substrings within the string.

Algorithm 6 LCP Construction

```
1:  function LCPConstruction(s: string, p: vector of integers) → vector of integers
2:      n ← length of s
3:      rank ← vector of length n, initialized with zeros
4:      for i ← 0 to n − 1 do                          ▷ Compute rank of each suffix
5:          rank[p[i]] ← i
6:      end for
7:      k ← 0
8:      lcp ← vector of length n − 1, initialized with zeros
9:      for i ← 0 to n − 1 do
10:         if rank[i] == n − 1 then
11:             k ← 0
12:             continue
13:         end if
14:         j ← p[rank[i] + 1]
15:         while i + k < n and j + k < n and s[i + k] == s[j + k] do
16:             k ← k + 1
17:         end while
18:         lcp[rank[i]] ← k
19:         if k then
20:             k ← k − 1
21:         end if
22:     end for
23:     return lcp
24: end function
```

3.6.1 Properties of the LCP Table

- **Size:** The LCP table has $N-1$ elements, where N is the length of the original string. Each element corresponds to the length of the longest common prefix between adjacent suffixes.

- **Value Range:** LCP values are non-negative integers, with a maximum value being the length of the shortest suffix in the pair.
- **Bounds:** The minimum LCP value is 0 (for the first and last suffixes) since there's no common prefix between them. The maximum LCP value is determined by the length of the shorter of the two suffixes being compared.
- **Relation to Suffix Array:** The LCP value for $SA[i]$ and $SA[i+1]$ tells you the length of the common prefix of the corresponding suffixes in the Suffix Array. It's also worth noting that the LCP values satisfy the "$LCP[i] \geq LCP[i-1] - 1$" property, implying that they don't decrease by more than one between consecutive elements.

3.6.2 Applications of the LCP Table

- **Repeated Substrings:** Longer LCP values indicate longer common substrings among adjacent suffixes. This can be used to find repeated substrings or substrings that occur multiple times in the original string.
- **Pattern Matching:** The LCP table can help accelerate substring and pattern matching algorithms. For instance, when searching for a pattern in the string, if you know that the LCP value between the pattern and the suffix in the array is greater than or equal to the length of the pattern, you can skip further comparisons.
- **Data Compression:** The LCP table is a crucial component in constructing data compression algorithms like the Burrows-Wheeler Transform (BWT), which is the basis for formats like BZip2.
- **Bioinformatics:** In DNA sequence analysis, the LCP table can help identify repetitive sequences and patterns in genomes.

3.6.3 Example

Consider the string "banana." The corresponding Suffix Array might be $[5, 3, 1, 0, 4, 2]$, and the LCP table could be $[0, 1, 3, 0, 0]$. This indicates that:

- The suffixes at index 1 (ana) and index 2 (nana) have an LCP of 1 (the letter "a").

- The suffixes at index 2 (nana) and index 3 (banana) have an LCP of 3 (the letters "ana").
- The rest of the suffixes don't share any common prefixes.

In summary, the Longest Common Prefix (LCP) table provides insights into the commonalities among consecutive suffixes in a Suffix Array. Its construction, properties, and applications make it a valuable data structure in various string-related tasks.

3.7 Treap Data Structure[4]

The Treap data structure is a combination of a binary search tree (BST) and a heap. It maintains elements in a way that combines properties of both a BST and a heap, providing balanced search and insertion operations while also ensuring that elements have a certain priority ordering. The term "Treap" is derived from the combination of "tree" (from BST) and "heap."

3.7.1 Structure

A Treap consists of nodes, each containing a key and a priority value. The keys in a Treap follow the rules of a binary search tree: the keys in the left subtree of a node are smaller than the key in the node, and the keys in the right subtree are greater. The priorities are assigned randomly or based on certain criteria (e.g., randomly generated or using an additional attribute).

3.7.2 Priority Property

The priority of a node follows the max-heap property: the priority of a parent node is greater than or equal to the priorities of its children. This ensures that the highest priority elements (max heap) or lowest priority elements (min-heap) are at the root of the Treap.

3.7.3 Rotation Operations

Rotation operations (left and right rotations) are performed to maintain the BST property while adjusting priorities to satisfy the heap property. These rotations are done to keep the tree balanced and to ensure that the priority order remains consistent.

3.7.4 Insertion

When inserting a new element into the Treap, it is initially added as a leaf node based on the BST rules. Then, the Treap is adjusted by performing rotations to maintain both the BST and heap properties. The rotation process involves comparing the priority of the new node with the priority of its parent and performing rotations as needed to satisfy the heap property.

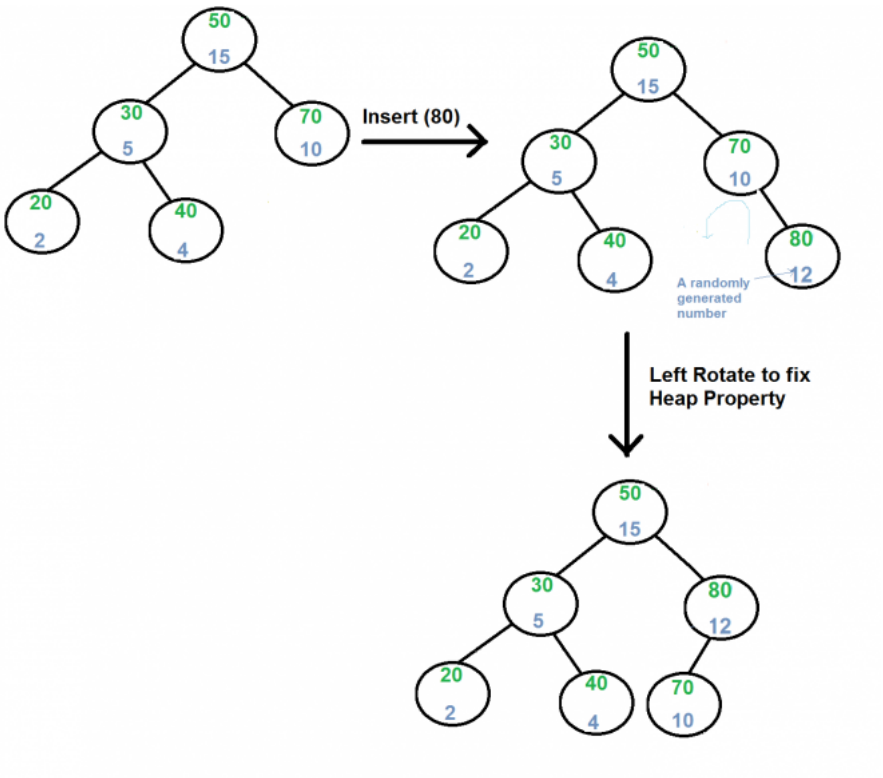

Figure 3.5: Insert in treap[4]

3.7.5 Deletion

To delete a node from the Treap, it is removed as in a normal BST. After the deletion, rotations may be performed to restore both the BST and heap properties.

3.7.6 Search

Searching for an element in a Treap is performed using the key comparison process similar to a BST.

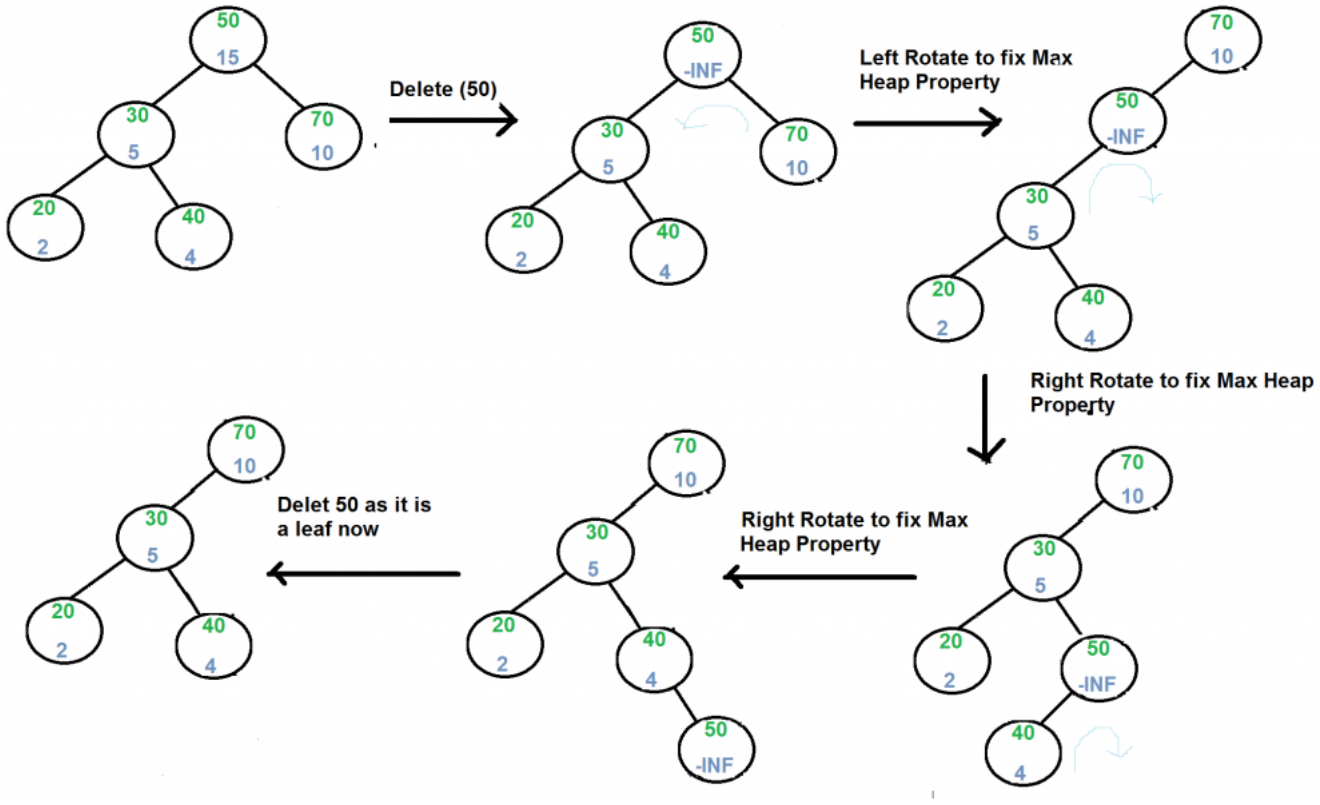

Figure 3.6: Delete in treap[4]

3.7.7 Priority Randomization

The randomness of priority assignments is a key feature of the Treap. This randomization ensures that the structure remains relatively balanced, preventing worst-case scenarios that can occur in traditional BSTs.

3.7.8 Balanced Structure

While Treaps are not guaranteed to have perfectly balanced structures like AVL trees or Red-Black trees, they tend to have good average-case performance due to the randomized priorities.

Chapter 4

Performance of existing model

All existing models that address the "All Pair Suffix-Prefix Matching Problem" operate in an offline manner. Consequently, when a new string is introduced, the entire model needs to be re-executed, resulting in significant computational overhead. To address this limitation, we conducted experiments on an existing model to assess its performance under dynamic scenarios involving the addition of strings.

Furthermore, we conducted experiments on both a brute-force approach and an optimized brute-force approach that incorporates hashing to a certain degree. For testing purposes, we use random DNA sequences as datasets. The random DNA sequence is generated through a random DNA sequence generator code which is mentioned below:

Algorithm 7 Generate Random Strings

```
Initialize mt19937_64 as rnd(chrono::steady_clock::now().time_since_epoch().count())
Let alphabet be the string "ATCGEFDHIJKLMNOPQRSBUVWXYZabcdefghijklmnopqrstu-
vwxyz0123456789"
function BuildRandomString(len, clen)
    ans ← ""
    for i ← 0 to len − 1 do
        ans ← ans + alphabet[rnd()  mod clen]
    end for
    return ans
end function
function StringVector(len, sz)
    Let ans be an empty vector of strings with size len
    for i ← 0 to len − 1 do
        cur_len ← sz
        ans[i] ← BuildRandomString(cur_len, 4)
    end for
    return ans
end function
procedure Main
    Initialize random number generator
    ans ← StringVector(300, 500)
    for it in ans do
        Print it followed by a newline
    end for
end procedure
```

For randomization purpose it uses mt19937. This is a Mersenne Twister based on the prime $2^{19937} - 1$. It's a much higher-quality Random Generator than rand(). More about mt19937 and why it is better has been described in the blog[19]. The build_random_string(int len, int clen) function generates random dna strings of size len and the string_vector(int len, int sz) function will take sz and len as a argument and will generate a vector of size equal to len that comprises of random DNA sequence of fixed length sz. The source that is used as a running model of three algorithms is provided in a GitHub repository [20]. In the subsequent tables, we present the running times of different algorithms across distinct scenarios, measured in milliseconds.

Number of Strings	Length of Every String	Bruteforce	Bruteforce + Hashing	Suffix Array
50	20	43	36	48
50	40	103	70	91
50	50	132	88	112
50	200	591	347	463
50	500	1645	870	1202
50	1000	3404	1735	2432
100	20	342	286	238
100	40	818	565	465
100	50	1053	704	553
100	200	4745	2788	2206
100	500	12758	6943	5841
100	1000	27261	13884	12604
300	20	9212	7753	3894
300	40	22009	15227	7461
300	50	29531	19297	9177
300	200	127664	75562	36952
300	500	347327	187633	93369
300	1000	1123451	586236	300754
1000	20	340753	286954	117768
1000	40	811652	570926	217567
1000	50	1045555	711234	278838
1000	200	4944345	2543125	1323112
1000	500	18745353	9645123	5123561
1000	1000	52345123	23124763	11129543

Table 4.1: Running time of the Algorithms in milliseconds(ms)

4.1 Observation

The naive brute-force approach consistently performs poorly across all scenarios when compared to the other two methods.

For a small number of strings (≤ 50), the "Bruteforce + Hashing" approach outperforms the "Suffix Array" approach. However, in the remaining cases, particularly with larger numbers of strings, the "Suffix Array" approach demonstrates better performance than the other two methods.

Nonetheless, as the number of strings and the size of each string increase, the running time experiences a significant increase. For instance, when there are 1000 strings, each of length 1000, the "Suffix Array" approach takes **11129.543 seconds**. This time complexity would further

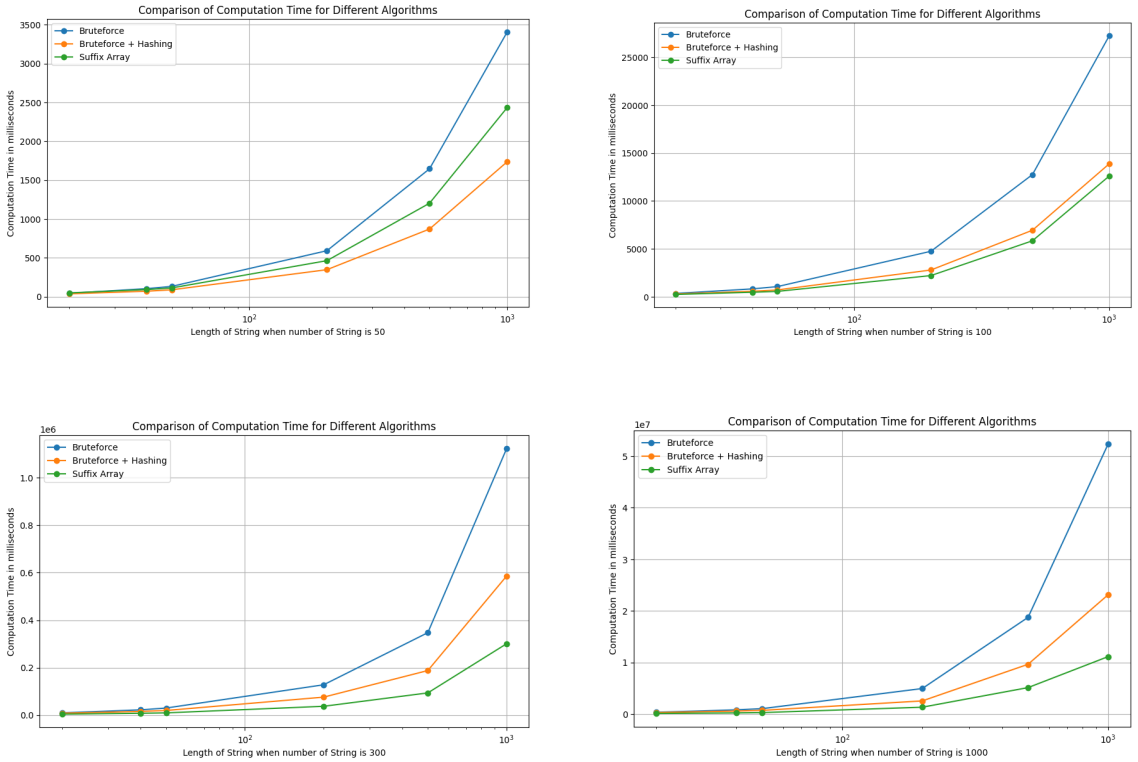

Figure 4.1: Performance of three algorithms

increase with the addition of more strings. As the complexity after incorporating a new string is $O(totLen + m^2)$, where $totLen$ represents the total length of all previous strings and m is the number of previous strings, the "Suffix Array" approach proves unsuitable for managing large and lengthy collections of strings. Our aim is to construct a model that operates with logarithmic complexity.

Chapter 5

Proposed model

5.1 Notation and defination

We use some definitions for our proposed model. Consider a string S with a length of $|S| = m$, comprised of symbols drawn from an ordered alphabet. We represent the i-th symbol of S as $S[i]$, where $1 \leq i \leq n$. $S[i, j]$ refers to a substring of S from position i to j, inclusive. Specifically, $S[1, j]$ denotes the prefix of S that concludes at position j, and $S[i, n]$ signifies the suffix of S commencing at position i, denoted as S_i. We employ the symbol to express the lexicographic order relationship between strings.

The Suffix Array of a string S, denoted as SA, is an array of integers ranging from 1 to m. This array ranks all the suffixes of S in lexicographic order. We denote the position of the suffix S_i in the Suffix Array as $\text{pos}(S_i)$.

The LCP-array, another crucial component, consists of integers that store the length of the longest common prefix (lcp) between consecutive suffixes in SA. $LCP[1]$ is set to 0, and for $1 < i \leq m$, $LCP[i]$ equals $lcp(S_{SA}[i], S_{SA}[i-1])$. Here, $lcp(u, v)$ represents the longest common prefix of strings u and v. Constructing both SA and the LCP-array can be achieved in linear time[**?**].

The range minimum query (RMQ) concerning the LCP array entails finding the smallest lcp value within a given interval of SA. We define $RMQ(i, j)$ as the minimum value of $LCP[k]$ for $i < k \leq j$. Remarkably, for a string S with length n and its corresponding LCP-array, the $lcp(S_{SA}[i], S_{SA}[j])$ is equivalent to $RMQ(i, j)$.

Let $S = \{S_1, S_2, \ldots, S_m\}$ be a collection of m strings. The generalized Suffix Array of S is the Suffix Array SA of the concatenated string $S = S^1\$_1\, S^2\$_2 \ldots S^m\$_m$, where each symbol $\$i$ is a distinct separator that does not occur in S and precedes every symbol in S, and $\$_i < \$_j$ if $i < j$. For a suffix $S_{SA}[i]$ of S, we denote the prefix of $S_{SA}[i]$ that ends at the first separator $\$_j$ by $S_{SA}[i]$. The total length of the generalized Suffix Array is $N = m + \sum_{l=1}^{m} |S_l|$.

To simplify the notation, we introduce the arrays STR and SA''. STR indicates the string in S from which a suffix came, formally $STR[i] = j$ if the suffix $S_{SA}[i]$ ends with symbol $\$_j$. SA'' holds the position of a suffix with respect to the string it came from (up to the separator), defined as $SA''[i] = k$ if $S_{SA}[i] = S_j^k\,\$j$. Taken together, STR and SA'' specify the order of all suffixes in S.

5.2 Methodology

Our model solution can solve the all-pair suffix-prefix matching problem when adding a new string. Let's assume that we already have m strings: $[S_1, S_2, S_3, \ldots, S_m]$, and we possess a grid named OV with a size of $m \times m$. In this grid, each cell (i, j) represents the maximum length of a suffix of string S_j that is also a prefix of string S_i. When a new string Sm' is introduced, we only need to update the m'-th row and m'-th column of the OV grid. Utilizing our existing model, we can avoid recalculating the entire grid, thereby significantly reducing complexity.

In contrast, using the conventional model would require a complete recalculation, leading to higher complexity. Specifically, the approximate complexity of adding a new string using our model is $O(\text{len} \cdot \log(\text{len})) + O(m \cdot \log(\text{tot\_len}))$, where len represents the length of the new string, m is the number of strings, and tot_len is the sum of the lengths of all strings plus m. On the other hand, the complexity of the existing model would be $O(\text{tot\_len} + m^2)$, which is considerably greater than that of our model.

Our model solution primarily relies on an online Suffix Array approach. To effectively implement the online Suffix Array, we employ data structures such as Treap, Suffix Array, LCP array, Segment Tree, BIT (Binary Indexed Tree), stack, and priority queue.

In our solution, we consider an existing grid OV with dimensions $m \times m$. Our objective is to add a new row and column to this grid. To accomplish this, we'll utilize the notations and definitions mentioned earlier.

We possess an existing Suffix Array that encompasses all the strings added thus far. Our approach involves continually appending new suffixes to this Suffix Array. To simplify our implementation of the online Suffix Array, we adopt a strategy where we add suffixes in reverse order, starting from the last character and working our way to the first character in the new string. For this purpose, we prepend each of these characters to the front of the string S, which is formed by concatenating all the previous strings while introducing a unique separator between them.

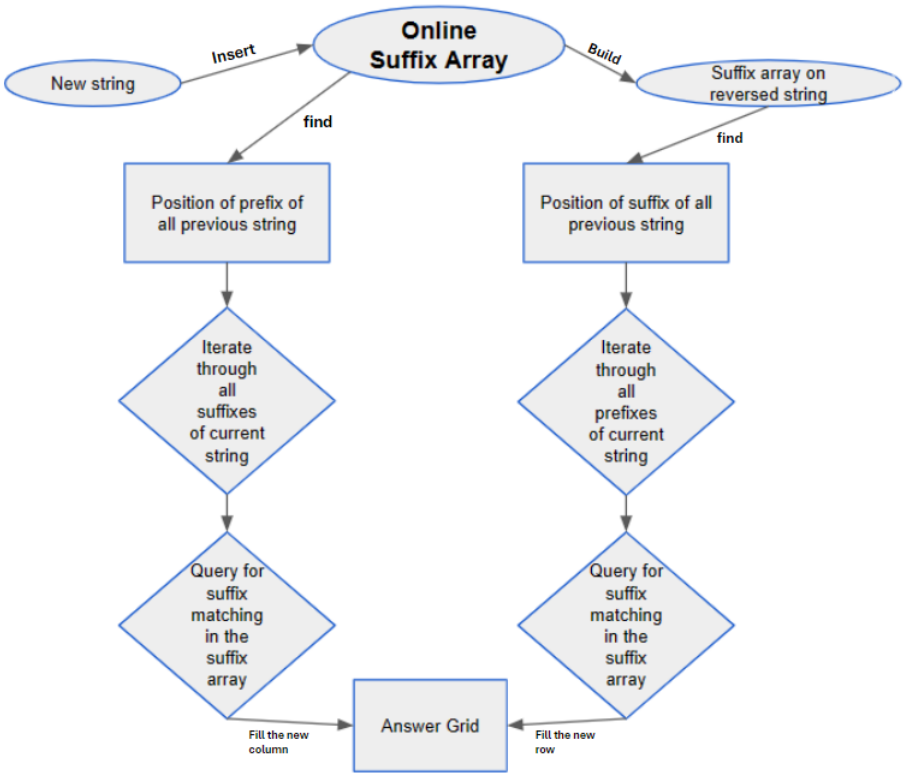

Figure 5.1: Our proposed model

By adding characters from the new string in reverse order to the beginning of string S, we are able to simplify our implementation process.

When incorporating a new suffix, several updates are required across multiple arrays: Suffix Array, LCP array, STR array, SA array, and SA' array. To facilitate these simultaneous updates, we employ a Treap data structure, which can handle complex updates and queries efficiently.

Additionally, a pre_array is maintained for each string, containing the positions of the respective strings in the Suffix Array. For every suffix within the Suffix Array, potentially qualifying as a candidate for a prefix match across a certain range within the Suffix Array, we monitor and adjust this range as new suffixes are added.

When a new suffix is introduced, its impact is confined to a small segment within the Suffix Array. Consequently, the complexity of updating the array remains logarithmic, given the nature of these incremental changes.

We approach the task by dividing it into two distinct parts. The first involves identifying the maximum matching prefix among all previous strings, where these prefixes are also suffixes of the current string. This part contributes to filling the new column in the grid. The second part entails finding the maximum matching suffix among all previous strings, where these suffixes serve as prefixes for the current string. This operation contributes to filling the new row in the grid. By addressing these **two parts** independently, we can effectively handle the problem and simplify the solution process.

5.2.1 First Task

In the first task, we will find the maximum matched suffix of the new string and the prefix of all previous strings. We use an online suffix array for this task. The following code is used for this task.

First, we add all suffixes in ascending order of their length to the suffix array. After that, we collect the positions of all previous strings in the suffix array and insert them into a set. Now we run a loop for all suffixes of the new string in descending order of their length.

For the current suffix, we find all the strings that are matched with this suffix. As later suffixes will be shorter than the current suffix, we delete all strings from the set that have a match. For this purpose, first, we find the position of the current suffix in the suffix array. Let pos be the position of the current suffix in the suffix array, and len be the length of that suffix. Now, a string having the position of pos_s in the suffix array has a match equal to LCP(pos_s, pos). Now, if this value is equal to len, we can say that the string has a match with the current suffix. For faster calculation, we use the following lemma:

Lemma 01: $\text{LCP}(i, j) \geq \text{LCP}(k, j)$, if $j > \max(i, k)$ and $k \leq i$.

Algorithm 8 First Task

```
1: function FirstTask
2:     for i ← cur_len − 1 to 0 do
3:         j ← 0
4:         cur_suffix_id ← sa.isa(j)
5:         it ← prefix_id.upper_bound({cur_suffix_id, −1})
6:         while it ≠ prefix_id.end() do
7:             (id, ind) ← ∗it
8:             lcp ← sa.query(new_string_add[ind], j)
9:             if lcp < i then
10:                break
11:            end if
12:            OV[ind][n − 1] ← i
13:            it ← prefix_id.erase(it)
14:        end while
15:        while it ≠ prefix_id.begin() do
16:            it ← it − 1
17:            (id, ind) ← ∗it
18:            lcp ← sa.query(new_string_add[ind], j)
19:            if lcp < i then
20:                break
21:            end if
22:            OV[ind][n − 1] ← i
23:            it ← prefix_id.erase(it)
24:        end while
25:    end for
26: end function
```

Lemma 02: LCP$(i, j) \geq$ LCP(k, j), if $j < \min(i, k)$ and $k \geq i$.

By using these lemmas, we divide all strings into two parts. One has a position in the suffix array less than pos, and the other has a position greater than pos. For the first set of strings, we go from a higher position to a lower one in the suffix array. Now, if a string doesn't have a match equal to len, all the following strings will have a lesser match. So we go through all strings, and if its match is equal to the suffix length, we add this length as the maximum matching between this string's prefix and the new string's suffix. We also remove this string from the set. We continue going through all remaining strings using the same process. If at any point some strings don't give a match equal to len, we stop there, as all the following strings will give a lesser match.

We follow the same process for the strings that have a higher position in the suffix array. Using this approach, we find a match between the new string's suffix and all previous strings' prefixes. This completes our first task.

5.2.2 Second Task

For the second task, we will find the maximum matched prefix of the new string and the suffix of all previous strings. We again use an online suffix array for this task. This time we consider all strings in reverse order. We use the following code for the second task.

First, we add all prefixes in ascending order of their length to the suffix array. After that, we collect the positions of all previous strings in the suffix array and insert them into a set. Now we run a loop for all prefixes of the new string in descending order of their length.

For the current prefix, we find all the strings that are matched with this prefix. As later prefixes will be shorter than the current prefix, we delete all strings from the set that have a match. For this purpose, first, we find the position of the current prefix in the suffix array. Let pos be the position of the current prefix in the suffix array, and len be the length of that prefix. Now, a string having the position of pos_s in the suffix array has a match equal to LCP(pos_s, pos). Now, if this value is equal to len, we can say that the string has a match with the current prefix. For faster calculation, we use the lemma mentioned in the first task.

By using these lemmas, we divide all strings into two parts. One has a position in the suffix array less than pos, and the other has a position greater than pos. For the first set of strings, we

Algorithm 9 Second Task

```
1: function SecondTask
2:     for i ← cur_len − 1 to 0 do
3:         j ← 0
4:         cur_suffix_id ← rev_sa.isa(j)
5:         it ← suffix_id.upper_bound({cur_suffix_id, −1})
6:         while it ≠ suffix_id.end() do
7:             (id, ind) ← ∗it
8:             lcp ← rev_sa.query(new_string_add[ind], j)
9:             if lcp < i then
10:                break
11:            end if
12:            OV[n − 1][ind] ← i
13:            it ← suffix_id.erase(it)
14:        end while
15:        while it ≠ suffix_id.begin() do
16:            it ← it − 1
17:            (id, ind) ← ∗it
18:            lcp ← rev_sa.query(new_string_add[ind], j)
19:            if lcp < i then
20:                break
21:            end if
22:            OV[n − 1][ind] ← i
23:            it ← suffix_id.erase(it)
24:        end while
25:    end for
26: end function
```

go through from a higher position to a lower position in the suffix array. Now, if a string doesn't have a match equal to len, all the following strings will have a lesser match. So we go through all strings, and if its match is equal to the prefix length, we add this length as the maximum matching between this string's suffix and the new string's prefix. We also remove this string from the set. We continue going through all remaining strings using the same process. If at any point some strings don't give a match equal to len, we stop there, as all the following strings will give a lesser match.

We follow the same process for the strings that have a higher position in the suffix array. Using this approach, we find a match between the new string's prefix and all previous strings' suffixes. This completes our second task.

5.3 Complexity Analysis

First task: For the first task, which fills the new column of the OV matrix, we use a Set that comprises the position of the prefix of all previous strings. After that, we iterate through all suffixes of the current string and find matches with previous strings. A prefix will be inserted and erased at most once. We also perform a binary search for each suffix to find the nearest prefix in the set. The total complexity is $\mathbf{O}(\mathbf{m} \cdot \log(\mathbf{m})) + \mathbf{O}(\text{len} \cdot \log(\mathbf{m}))$.

Where len is the length of the new string and m is the number of strings.

Second task: For the second task, which fills the new row of the OV matrix, we use a Set that comprises the position of the suffix of all previous strings. After that, we iterate through all prefixes of the current string and find matches with previous strings. A suffix will be inserted and erased at most once. We also perform a binary search for each prefix to find the nearest suffix in the set. The total complexity is $\mathbf{O}(\mathbf{m} \cdot \log(\mathbf{m})) + \mathbf{O}(\text{len} \cdot \log(\mathbf{m}))$.

Where len is the length of the new string and m is the number of strings.

Chapter 6

Experimental Analysis

In the previous chapter, following the execution of the Suffix Array model, we observed a significant overhead in accommodating new strings. The complexity associated with handling new strings using the Suffix Array approach is $\mathbf{O(totLen + m^2)}$, whereas our model boasts a complexity of $\mathbf{O(len * log(m) + m * log(totLen))}$, where $totLen$ represents the summation of the lengths of all previous strings, m is the number of previous strings, and len corresponds to the length of the new string. We compare our model with the previous two models: the suffix array and Bruteforce + Hashing. For testing purposes, we use the code[21] mentioned above. In the following table, we show the runtime of our algorithms and the previous models. 6.1

Number of Strings	Length of Every String	Bruteforce + Hashing	Suffix Array	Our Model
50	20	36	48	10
50	40	70	91	20
50	50	88	112	26
50	200	347	463	114
50	500	870	1202	311
50	1000	1735	2432	677
100	20	286	238	26
100	40	565	465	48
100	50	704	553	60
100	200	2788	2206	254
100	500	6943	5841	697
100	1000	13884	12604	1490
300	20	7753	3894	137
300	40	15227	7461	219
300	50	19297	9177	260
300	200	75562	36952	936
300	500	187633	93369	2495
300	1000	586236	300754	5608
1000	20	286954	117768	1188
1000	40	570926	217567	1521
1000	50	711234	278838	1685
1000	200	2543125	1323112	4512
1000	500	9645123	5123561	11664
1000	1000	23124763	11129543	25658

Table 6.1: Running time of the Algorithms in milliseconds(ms)

We observed that the runtime of our code is less than the other two in almost every case. This indicates significantly faster processing time compared to the previous models. Hence, we can confidently state that our model outperforms the existing models when dealing with dynamic string addition.

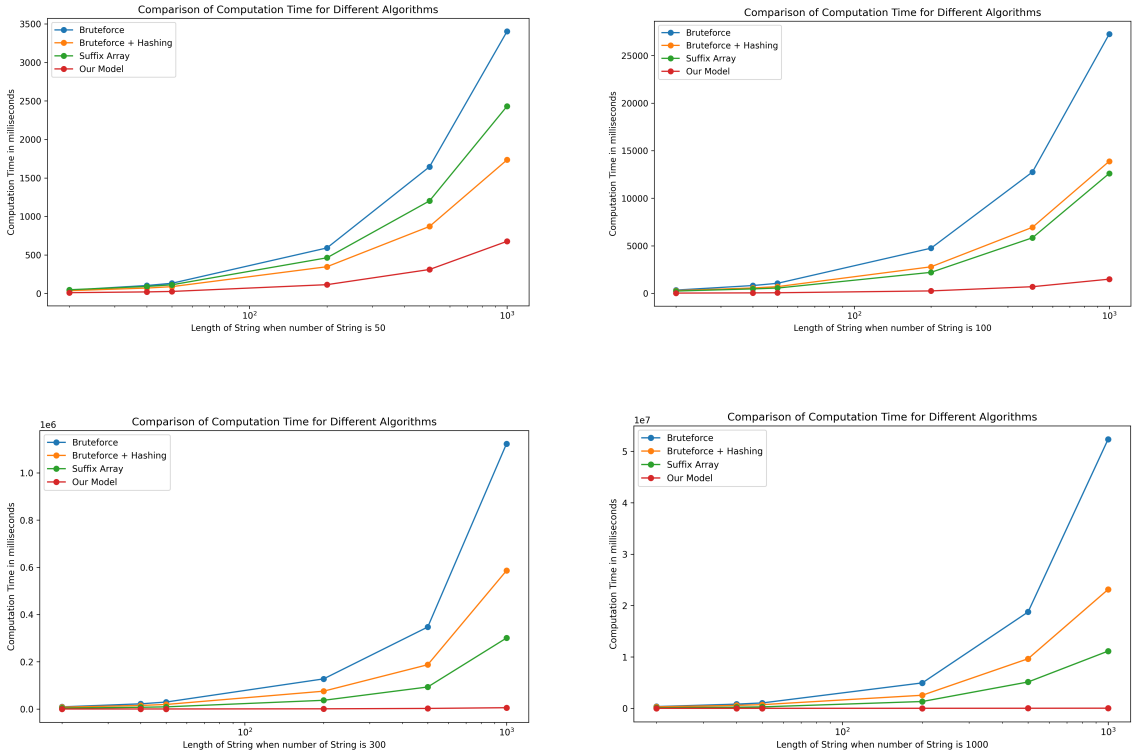

Figure 6.1: Performance of existing algorithms vs our model.

Chapter 7

Conclusion

This thesis addresses the limitations of the traditional all-pair-suffix-prefix matching problem by incorporating the handling of the insertion of new strings. Through the implementation of an innovative algorithmic approach, we have established the groundwork for efficient calculations of all-pair suffix-prefix matches with existing strings, while also providing seamless integration for the insertion of new strings.

We have successfully implemented algorithms that overcome previous limitations. However, our model introduces some overhead as we utilized heavy data structures, resulting in a higher constant factor. Consequently, the operation time is not precisely logarithmic; it is actually 3-5 times higher.

7.1 Future Works

We anticipate that future researchers will enhance the operational complexity to a much lower range. It would be particularly significant if it becomes feasible to reduce the complexity below the logarithmic range, achieving a constant factor.

References

[1] J. Lim and K. Park, “A fast algorithm for the all-pairs suffixâprefix problem,” *Theoretical Computer Science*, vol. 698, pp. 14–24, 2017, algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397517305510

[2] We. (2024, August) Final solution idea. Accessed on 26 - 02 - 2024. [Online]. Available: https://github.com/AlfehSani/4_2_Thesis

[3] Page - Algorithms for Competitive Programming, “Suffix array - algorithms for competitive programming,” August 2023, accessed on Day Month Year. [Online]. Available: https://cp-algorithms.com/string/suffix-array.html

[4] cp algorithms, “Treap - competitive programming algorithms,” 2023, website. [Online]. Available: https://cp-algorithms.com/data_structures/treap.html

[5] Page - Algorithms for Competitive Programming, “Sparse table - algorithms for competitive programming,” August 2023, accessed on Day Month Year. [Online]. Available: https://cp-algorithms.com/data_structures/sparse-table.html

[6] ——, “Main page - algorithms for competitive programming,” January 2023, accessed on August 20, 2023. [Online]. Available: https://cp-algorithms.com/

[7] cp algorithms, “Segment tree - competitive programming algorithms,” 2023, website. [Online]. Available: https://cp-algorithms.com/data_structures/segment_tree.html

[8] Page - Algorithms for Competitive Programming, “Longest common prefix of two substrings with additional memory - algorithms for competitive programming,” August 2023, accessed

on Day Month Year. [Online]. Available: https://cp-algorithms.com/string/suffix-array.html#longest-common-prefix-of-two-substrings-with-additional-memory

[9] "Binary search tree - geeksforgeeks," https://www.geeksforgeeks.org/binary-search-tree-data-structure/, (Accessed on 02/24/2024).

[10] "[tutorial] dynamic suffix arrays - codeforces," https://codeforces.com/blog/entry/93042, (Accessed on 02/22/2024).

[11] W. H. Tustumi, S. Gog, G. P. Telles, and F. A. Louza, "An improved algorithm for the all-pairs suffixâprefix problem," *Journal of Discrete Algorithms*, vol. 37, pp. 34–43, 2016, 2015 London Stringology Days and London Algorithmic Workshop (LSD LAW). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866716300053

[12] D. Gusfield, G. M. Landau, and B. Schieber, "An efficient algorithm for the all pairs suffix-prefix problem," *Information Processing Letters*, vol. 41, no. 4, pp. 181–185, 1992.

[13] E. Ohlebusch and S. Gog, "Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem," *Information Processing Letters*, vol. 110, no. 3, pp. 123–128, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019009003275

[14] M. Haj Rachid, Q. Malluhi, and M. Abouelhoda, "Using the sadakane compressed suffix tree to solve the all-pairs suffix-prefix problem," *BioMed research international*, vol. 2014, p. 745298, 04 2014.

[15] W. H. Tustumi, S. Gog, G. P. Telles, and F. A. Louza, "An improved algorithm for the all-pairs suffixâprefix problem," *Journal of Discrete Algorithms*, vol. 37, pp. 34–43, 2016, 2015 London Stringology Days and London Algorithmic Workshop (LSD LAW). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866716300053

[16] G. Loukides and S. P. Pissis, "All-pairs suffix/prefix in optimal time using aho-corasick space," *Information Processing Letters*, vol. 178, p. 106275, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019022000321

[17] Page - Algorithms for Competitive Programming. (2023, August) Range minimum query (rmq) - algorithms for competitive programming. Accessed on Day Month Year. [Online]. Available: https://cp-algorithms.com/sequences/rmq.html

[18] “The c++ standard template library (stl),” https://www.geeksforgeeks.org/the-c-standard-template-library-stl/, (Accessed on 02/26/2024).

[19] “Don’t use rand(): a guide to random number generators in c++ - codeforces,” https://codeforces.com/blog/entry/61587, (Accessed on 02/24/2024).

[20] We. (2023, August) Initail model. Accessed on 20 - 08 - 2023. [Online]. Available: https://github.com/AlfehSani/4_1_Thesis

[21] ——. (2023, August) Random dna sequence generator. Accessed on 20 - 08 - 2023. [Online]. Available: https://github.com/AlfehSani/4_1_Thesis/blob/main/generator.cpp