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

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

MAHBUBUL HASAN
Reg. No.: 2018331123
$4^{th}$ year, $1^{st}$ Semester

ALFEH SANI
Reg. No.: 2018331119
$4^{th}$ year, $1^{st}$ Semester

Department of Computer Science and Engineering

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

21st August, 2023

# 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

| | |
|---|---|
| Mahbubul Hasan | Alfeh Sani |
| Reg. No.: 2018331123 | Reg. No.: 2018331119 |
| $4^{th}$ year, $1^{st}$ Semester | $4^{th}$ year, $1^{st}$ Semester |

Department of Computer Science and Engineering

**Supervisor**

DR. HUSNE ARA CHOWDHURY

Associate Professor

Department of Computer Science and Engineering

21st August, 2023

# 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: 21st August, 2023

# 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 21st August, 2023, 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 | Dr. Sadia Sultana | Dr. Husne Ara Chowdhury |
| Professor | Associate 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," which involves finding 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 need 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 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 model.

**Keywords:** Suffix Array, Treap(cartesian tree), APSP, RMQ, Segment Tree [1]

# 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 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 matching for a new string $s^{'}$, which is not present in the set, we have to rebuild the whole model and need to compare with every string that was present in the set. Thus the time complexity increases by O(n) for every new insertion, where n is size of 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 APSP matching problem, that presented new space-efficient algorithms using compressed data structures. [7]

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 find prefix-suffix matching which is good compare to the older algorithms.

## 1.1 Motivation

**Algorithmic Research and Optimization:** Insertion of a new string in APSP matching problem, it 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 need 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.

# 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.[8] 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.[9] 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.[10] 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.[11] 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[12].

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[1] 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 [2]

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:

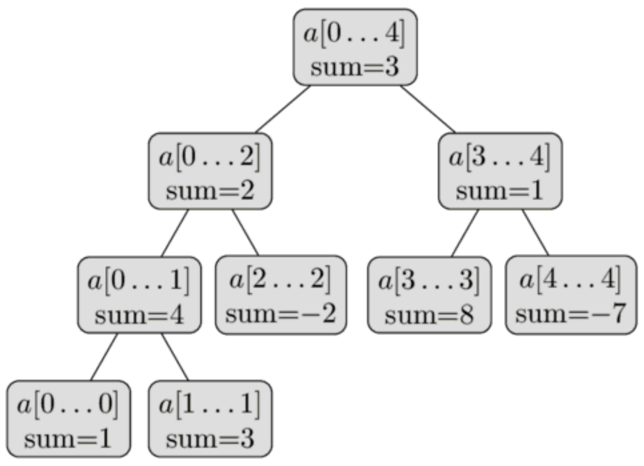

Figure 3.1: Segment Tree

### 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.

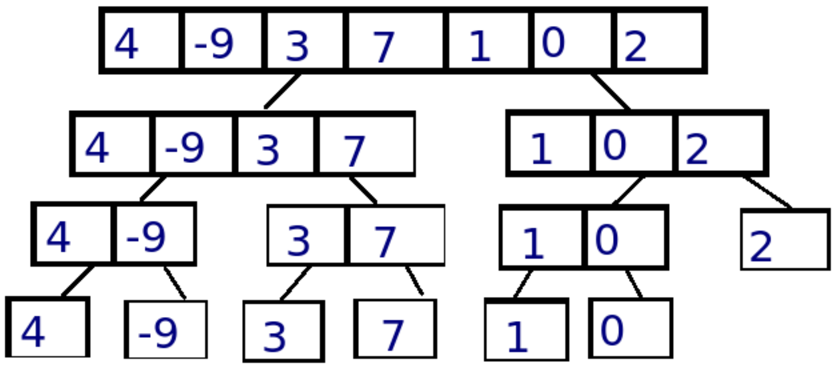

Figure 3.2: Segment Tree construction

### 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.

### 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: [3]

To perform range queries such as **Range Minimum Query** [13] 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

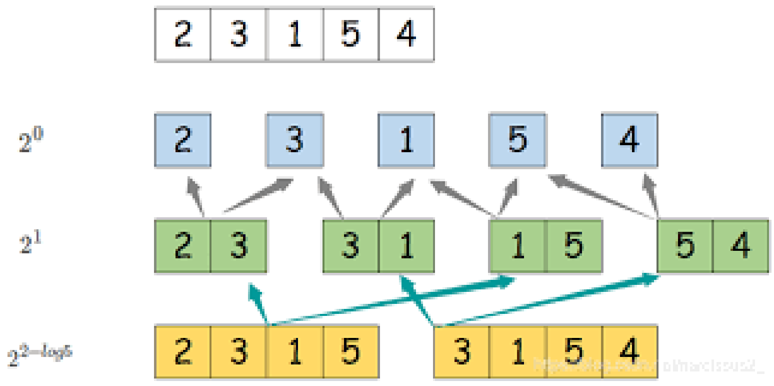

Figure 3.3: Sparse Table Example

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.

## 3.3 Suffix Array [4]

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 suffix according to lexicographical order.

### 3.3.2 Construction[5]

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.

### 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 The Longest Common Prefix (LCP) Table [5]

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.

### 3.4.1 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.

### 3.4.2 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.

### 3.4.3 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.

## 3.5 Treap Data Structure[6]

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."

### 3.5.1 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).

### 3.5.2 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.

### 3.5.3 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.

### 3.5.4 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.

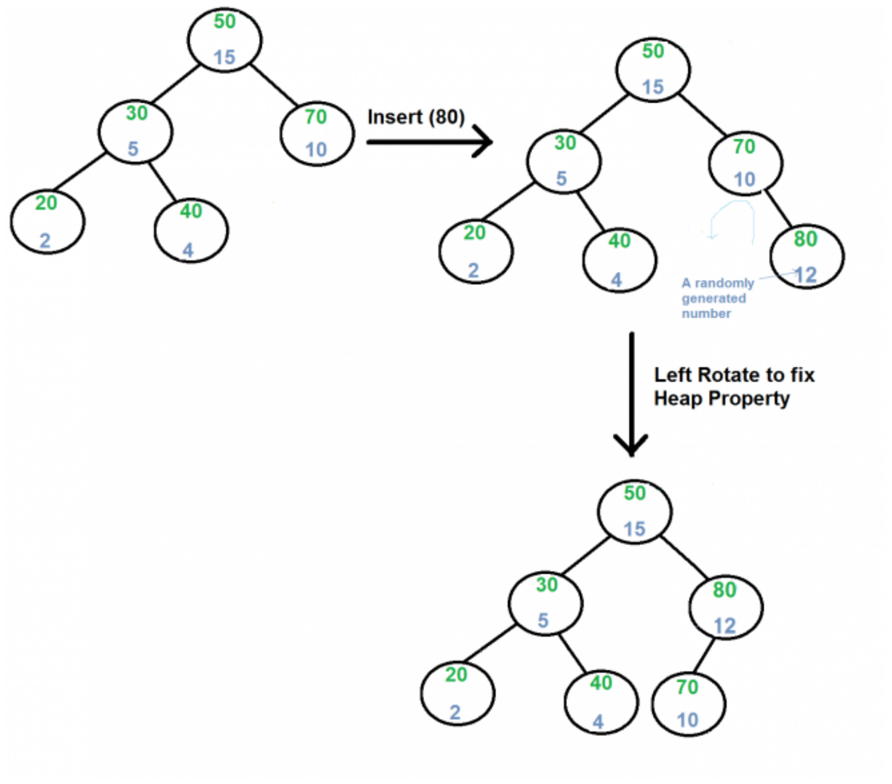

Figure 3.4: Insert in treap[6]

### 3.5.5 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.

### 3.5.6 Search

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

### 3.5.7 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.

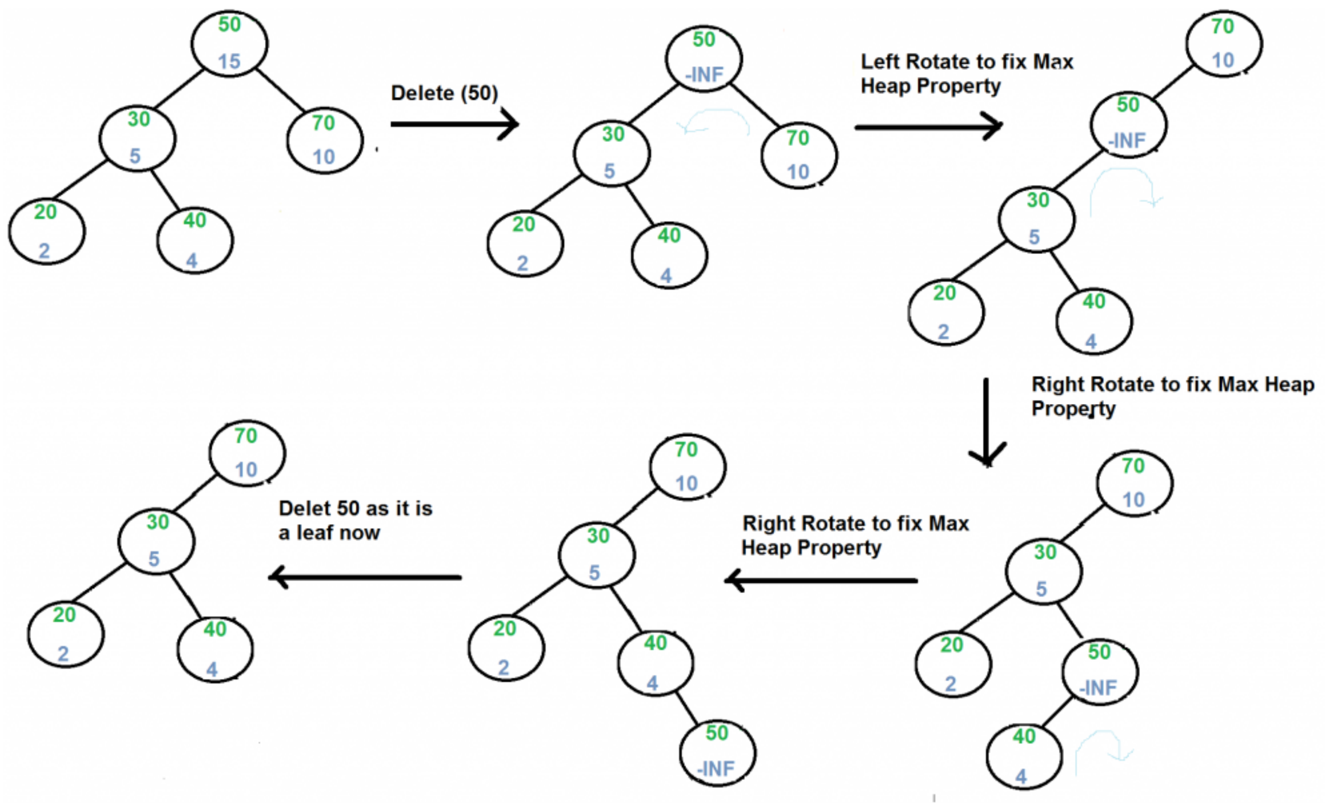
50
15
30
5
70
10
20
2
40
4
Delete (50)
50
-INF
30
5
70
10
20
2
40
4
Left Rotate to fix Max Heap Property
70
10
50
-INF
30
5
20
2
40
4
Right Rotate to fix Max Heap Property
70
10
30
5
20
2
50
-INF
40
4
Right Rotate to fix Max Heap Property
70
10
30
5
20
2
40
4
50
-INF
Delet 50 as it is a leaf now
70
10
30
5
20
2
40
4

Figure 3.5: Delete in treap[6]

### 3.5.8 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 by a code[14]. The source that is used as a running model of three algorithms is provided in a GitHub repository[15].

In the subsequent tables, we present the running times of different algorithms across distinct scenarios, measured in milliseconds.

## 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 ($\leq 50$), 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

| 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)

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 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.

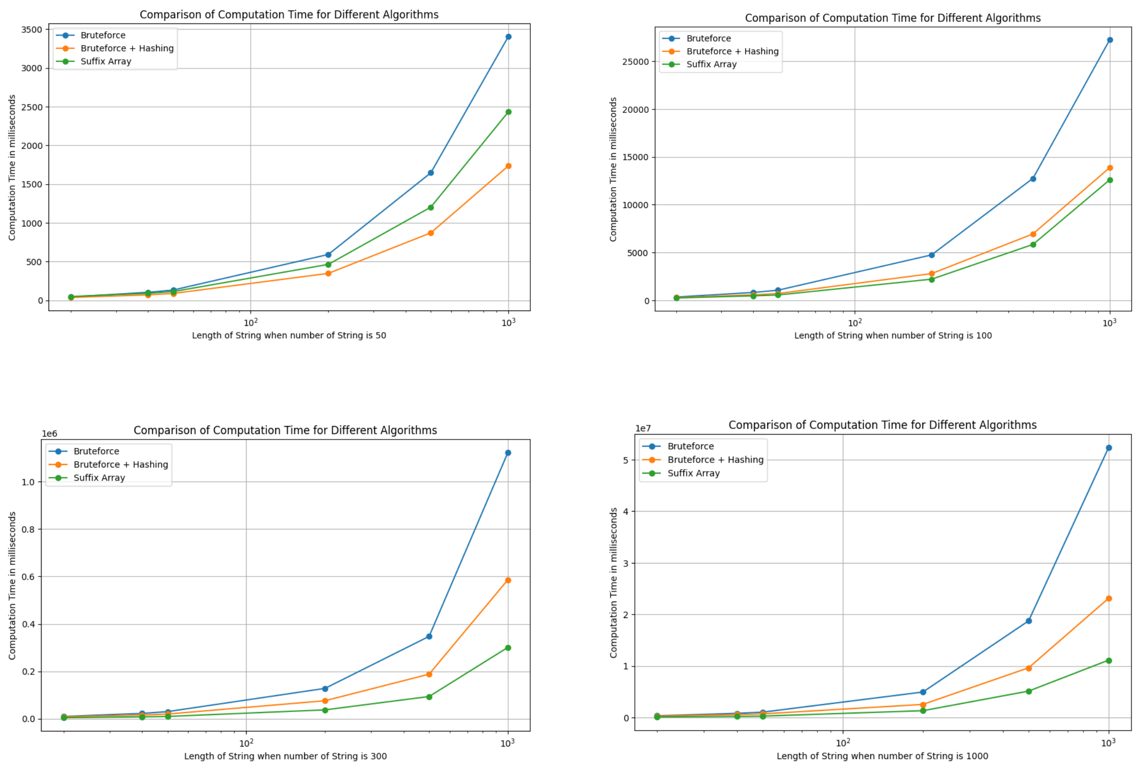

Figure 4.1: Performance of three algorithms

# 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[16].

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.

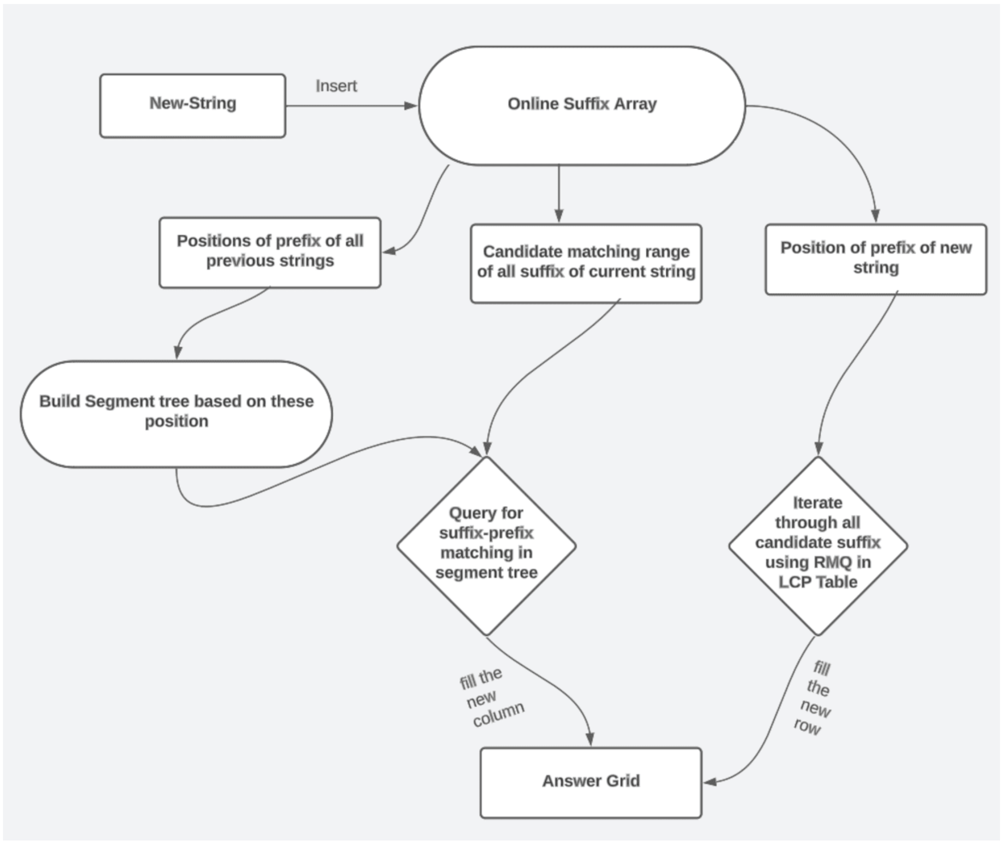
New-String
Insert
Online Suffix Array
Positions of prefix of all previous strings
Candidate matching range of all suffix of current string
Position of prefix of new string
Build Segment tree based on these position
Query for suffix-prefix matching in segment tree
Iterate through all candidate suffix using RMQ in LCP Table
fill the new column
fill the new row
Answer Grid

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

To tackle the first task, we create an array that holds the positions of all suffixes in the new string, ordered in descending order of their lengths. Additionally, we maintain another array that holds the positions of prefixes from all previous strings. We iterate through the first array and identify all prefixes that have a matching substring of the same length as the corresponding suffix. This information is then used to update the respective cell in the $OV$ grid. When we identify a matching prefix, we are aware that any upcoming suffixes will have shorter matches, enabling us to remove those prefixes with matches from consideration.

The primary challenge revolves around establishing matches between a suffix and the remaining prefixes. To address this, we leverage insights derived from the Suffix Array and LCP array. Here, we define a "suffix match" as a match with another string where the length of the matched portion is equal to the suffix length. Within the Suffix Array, a suffix has matches with certain strings over a contiguous range within the array. If a suffix of length $l$ occupies position $i$ in the Suffix Array, it will have matches within the range $[L, R]$ if $L \leq i \leq R$, and the minimum $LCP$ value in the range $[LCP[L+1], LCP[L+2], \ldots, LCP[R]]$ is equal to $l$.

To simplify this process, we precalculate candidate ranges $(L, R)$ for all suffixes in the new string. With these candidate ranges in place, the next step involves maintaining a Segment Tree containing the positions of prefixes from all previous strings. Using a compressed Segment Tree is essential to manage complexity effectively. By utilizing this Segment Tree, we can readily identify all prefixes that match the current suffix and adjust the Segment Tree accordingly to achieve the desired updates. This approach allows us to efficiently address the task at hand.

### 5.2.2 Second Task

The second task is indeed more intricate than the first one. To address it effectively, we can rely on a few key observations based on the lemmas provided earlier. Let's break down the approach into steps:

1. Find the position of the prefix of the newly added string in the Suffix Array.
2. Recognize that all matched suffixes will either come before this prefix or directly follow it. For suffixes that follow it directly, their matching length equals the length of the new string. Iterate through these directly following suffixes to determine their matching lengths.
3. Now, the focus is on identifying matching suffixes of other strings that precede the prefix of the new string. Let the position of the prefix be denoted as $i$. A suffix with a position $j$ and length $len$ (where $j < i$) will match with prefix $i$ if and only if $\min(LCP[j+1], LCP[j+2], \ldots, LCP[i]) = len$. This value essentially represents the maximum matching prefix between strings at positions $i$ and $j$.
4. To optimize time complexity, it's crucial to consider only those suffixes that are candidates for matching. Notably, iterating from $i$ to $k$ (where $k < i$) would result in decreased matching. Therefore, we need to traverse only through the candidate suffixes.
5. To further refine this approach for optimal time complexity, let $x$ represent $LCP[i]$, where $i$ is the current position. The next candidate suffix with position $k$ must satisfy the condition $LCP[k+1] \leq x$. To find the next largest $k$ that fulfils this condition, we can employ a Range Minimum Query (RMQ)[13] data structure.

By adopting this approach, we significantly reduce unnecessary iterations and focus solely on the candidate suffixes that have the potential to match with the prefix. This leads to a notable reduction in complexity and an efficient solution to the second task.

### 5.2.3 Complexity analysis

**First task:** For the first task, which fills the new column of the $OV$ matrix, we use a Segment Tree that comprises the position of the prefix of all previous strings. After that, we query in the candidate range for all suffixes of the current string in the Segment Tree. So total complexity will be $\mathbf{O(len * log(m)}$, 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 only check the candidate suffixes of the previous strings. As we mainly focused on DNA sequences, that is mainly random string, the number of candidate suffixes should be $\mathbf{O(m)}$, where m is the number of previous strings. To jump to the next candidate strings, we use RMQ and Binary search, where complexity for a jump is $\mathbf{O(log(totLen))}$, $totLen$ = total length of all previous strings + number of previous strings. So total complexity is $\mathbf{m * log(totLen)}$.

### 5.2.4 Comparison

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. In comparison to the previous model of the suffix array, which takes approximately $\mathbf{11129.543seconds}$ (approximately $\mathbf{200minutes}$) to process 1000 strings of size 1000, our model would only require $\mathbf{42.5992seconds}$. This indicates a significantly faster processing time in comparison to the previous model. Hence, we can confidently state that our model outperforms the existing model when dealing with dynamic string addition.

# Chapter 6

# Conclusion

This thesis addressed the limitation of traditional all-pair-suffix-prefix matching problem by handling the insertion of new strings. By proposing an innovative algorithmic approach, we have laid the foundation for efficient calculations of all-pair suffix-prefix matches with existing strings, while also offering seamless integration for the insertion of new strings.

While our proposed algorithm holds promising potential, it is important to acknowledge the work is at initial stage of development. Till now we have evaluated the performance of existing models and as mentioned, our algorithm's performance has yet to be validated through implementation. We intend to undertake this task in the upcoming semester.

# References

[1] Page - Algorithms for Competitive Programming, “Main page - algorithms for competitive programming,” January 2023, accessed on August 20, 2023. [Online]. Available: https://cp-algorithms.com/

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

[3] 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

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

[5] ——, “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

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

[7] 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

[8] 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.

[9] 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

[10] 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.

[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] 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

[13] 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

[14] We. (2023, August) Random dna sequence generator. Accessed on 20 - 08 - 2023. [Online]. Available: https://github.com/2018331119/4_1_Thesis/blob/main/generator.cpp

[15] ——. (2023, August) Soruce code for the algorithms. Accessed on 20 - 08 - 2023. [Online]. Available: https://github.com/2018331119/4_1_Thesis/tree/main

[16] D. K. Kim, J. S. Sim, H. Park, and K. Park, "Constructing suffix arrays in linear time," *Journal of Discrete Algorithms*, vol. 3, no. 2-4, pp. 126–142, 2005.