WAB

Provadis School of International Management and Technology

Comparing Suffix Automata Against Suffix Arrays For Longest Common Substring Queries

Rubin Chempananickal James

rubin.chempananickaljames@stud-provadis-hochschule.de

Matriculation Number: D876

Department: Information Technology

Module: Algorithmen und Datenstrukturen

Reviewer: Prof. Dr. Volker Scheidemann

February 26, 2026

Abstract

The objective of this paper was to compare the performance of two algorithms, the Suffix Automaton and the Enhanced Suffix Array, for solving the Longest Common Substring problem **between two strings** in a pure Python implementation.

The algorithms were tested on multiple scenarios, from best case to worst case, and on multiple string lengths, from 100 to 10,000 characters, with multiple runs for each scenario and string length to ensure statistical significance of the results. Both the build time and the query time were measured, as well as the memory used both for building the data structure and for querying it.

The results showed that **so long as one is programming in pure Python, and so long as only two strings are involved**, an Enhanced Suffix Array offers practically zero advantages over a Suffix Automaton. The Suffix Automaton was substantially faster to build and to query in all but one scenario, and although the Enhanced Suffix Array had an overall smaller index size, this would do very little in real-life memory-constrained applications, as its peak memory usage was either comparable to the Suffix Automaton or only barely smaller in all scenarios and string lengths. Not to mention that the Suffix Automaton is much simpler to implement.

This is potentially due to the nature of Python, as it is a high level interpreted language where the memory management is abstracted away from the programmer, leading to even simple data types like integers and lists having a much higher memory overhead than in lower level languages. Due to Python's memory management, the Enhanced Suffix Array would also lose its cache locality, which is the main reason for its compactness in lower level languages like C.

Abstrakt

Das Ziel dieser Arbeit war es, die Leistung von zwei Algorithmen, dem Suffix Automaton und dem Enhanced Suffix Array, bei der Lösung des Longest Common Substring-Problems ***zwischen zwei Zeichenketten (Strings)*** *in einer reinen Python-Implementierung zu vergleichen.*

Die Algorithmen wurden in verschiedenen Szenarien getestet, von besten bis zu schlechtesten Fällen, und auf verschiedenen Stringlängen, von 100 bis 10.000 Zeichen, mit mehreren Durchläufen für jedes Szenario und jede Stringlänge, um die statistische Signifikanz der Ergebnisse sicherzustellen. Sowohl die Aufbauzeit als auch die Abfragezeit wurden gemessen, ebenso wie der für den Aufbau der Datenstruktur und für die Abfrage verwendete Speicher.

Die Ergebnisse zeigten, dass ***solange man in reinem Python programmiert und nur zwei Zeichenketten beteiligt sind****, ein Enhanced Suffix Array praktisch keine Vorteile gegenüber einem Suffix Automaton bietet. Der Suffix Automaton war wesentlich schneller im Aufbau und in der Abfrage in allen bis auf einer der Szenarien, und obwohl der Enhanced Suffix Array insgesamt eine kleinere Indexgröße hatte, würde dies in realen, speicherbeschränkten Anwendungen nur sehr wenig ausmachen, da sein Spitzen-Speicherverbrauch in allen Szenarien und Stringlängen entweder vergleichbar mit dem des Suffix Automatons oder nur ein wenig kleiner war. Auch von Vorteil ist, dass der Suffix Automaton viel einfacher zu implementieren ist.*

Dies liegt vermutlich an der Natur von Python, da es eine hochstufige interpretierte Sprache ist, bei der die Speicherverwaltung von der Programmierer/in abstrahiert wird, was dazu führt, dass selbst einfache Datentypen wie Ganzzahlen und Listen einen viel höheren Speicher-Overhead haben als in kompilierten Sprachen. Aufgrund der Speicherverwaltung von Python würde der Enhanced Suffix Array auch seine Cache-Lokalität verlieren, was der Hauptgrund für seine Kompaktheit in Sprachen wie C ist.

Contents

List of Figures

List of Tables

Glossary

base The building blocks of DNA and RNA, represented by the letters A, C, G, T (for DNA) and A, C, G, U (for RNA). 5

DNA Deoxyribonucleic Acid, the molecule that carries genetic information in living organisms. 1

RNA Ribonucleic Acid, a molecule that is transcribed from DNA and which then gets translated into proteins. 1

sentinel A unique character used to separate concatenated strings in a suffix array or suffix tree, which is lexicographically smaller than any other character in the strings. Usually characters like \0 (null character), $, or # are used as sentinels, as they are ahead of all other characters in the ASCII table. 3

1 Introduction

1.1 Background

The Longest Common Substring (LCS) problem is a fundamental problem in computer science, with applications in multiple domains, bioinformatics being one of the most prominent. This is due to the fact that DNA and RNA sequences are usually encoded as strings for bioinformatics workflows, in a format called FASTA, originally described by Pearson and Lipman in 1985[1].

The LCS problem is defined as follows:

> Given two strings S and T, each of length at most n, the longest common substring (LCS) problem is to find a longest substring common to S and T.[2]

For example, given the strings "AG**CTAGC**" and "T**CTAGC**TA", the longest common substring is "CTAGC", which has a length of 5. The LCS problem can be solved using various algorithms, such as dynamic programming, suffix trees, suffix arrays, and suffix automata, each with different time and space complexities.

Of particular interest are the Suffix Automaton (SAM) (also known as Directed Acyclic Word Graph (DAWG)) and the Suffix Array (SA) with Longest Common Prefix (LCP) Array (also known as an Enhanced Suffix Array (ESA)), which both yield the search result in linear time, but differ in their construction time and space requirements.

Both of these data structures are derived from the Suffix Tree, the first linear time and space data structure for solving the LCS problem, which was introduced by Peter Weiner in 1973[3] as "bi-trees", although the uncompressed version he introduced is currently referred to as a *trie* in order to differentiate it from the modern definition of a Suffix Tree, which corresponds to the compressed version as described by McCreight in 1976[4].

1.1.1 Suffix Tree

A (compressed) Suffix Tree is constructed from a string S using an algorithm, originally called Algorithm M, mapping a finite string S of characters into an auxiliary index to S in the form of a digital search tree T whose paths are the suffixes of S, and whose terminal nodes correspond uniquely to positions within S[4].

As the Suffix Automaton and the Enhanced Suffix Array are both further refinements of the Suffix Tree, a Suffix Tree implementation will not be included in the comparisons done in this paper.

1.1.2 Suffix Automaton

A Suffix Automaton (SAM) is a data structure that represents all the substrings of a given string in a compact way.

> An efficient and compact data structure for representing a full index of a set of strings is a Suffix Automaton, a minimal deterministic automaton representing the set of all suffixes of a set of strings. Since a substring is a prefix of a suffix, a suffix automaton can be used to determine if a string x is a substring in time linear in its length *O(|x|)*, which is optimal.[5]

The SAM was first introduced by Blumer et al. in 1985[6], and its use in solving the LCS problem has been explored in various works, such as in the book *Text Algorithms* by Crochemore and Rytter, in chapter 6[7].

A notable feature of the Suffix Automaton is that it can be constructed with just one string, and in fact, the same automaton can then be reused to find the Longest Common Substring of that string and any other string, by traversing the automaton with the second string and keeping track of the longest match found. The same automaton can therefore be used to find the LCS of the first string and any number of other strings, without having to reconstruct it every time.

1.1.3 Enhanced Suffix Array

> The generic name *enhanced suffix array* stands for data structures consisting of the suffix array and additional tables.[8]

In this implementation, a Enhanced Suffix Array (ESA) consists of a Suffix Array (SA) and an Longest Common Prefix (LCP) Array, which is all that's necessary to solve the

LCS problem in linear time because the LCS can be solved by finding the maximum value in the LCP Array corresponding to suffixes (from the SA) that belong to different strings.

A SA is an array of integers that represent the starting positions of the suffixes of a string in lexicographical order, while the LCP Array is an array of integers that represent the length of the longest common prefix between adjacent suffixes in the SA[8].

Currently, there are algorithms to construct both the SA and the LCP Array in *O(m+n)* time for two strings S and T of lengths m and n respectively, such as the Suffix Array with Induced Sorting (SA-IS) algorithm for the SA[9] and Kasai's algorithm for the LCP Array[10].

A notable feature of the ESA is that it can be constructed with more than two strings at once, by concatenating the strings together with unique delimiters (sentinels), and then constructing the ESA for the concatenated string. This allows for finding the LCS of all the strings at once.

1.2 Motivation

Although there have been many papers comparing either the SAM or the ESA to the Suffix Tree[5,6,7,8,11], there is a lack of direct comparisons between these two data structures, even though they both fundamentally solve the same problem using the same underlying principle of representing the suffixes of a string in a compact way.

Moreover, there is a lack of practical comparisons between these two structures in the context of specific programming languages, even though the obtuse memory management of languages like Python[12] could potentially lead to real-world results not matching up with theoretical expectations, especially when it comes to space usage.

1.3 Research Question

The primary research question of this paper is:

> How do the SAM and the ESA compare in terms of time and space complexity when solving the Longest Common Substring problem **between two strings** in an everyday programming context?

Python[12] was chosen as the programming language for this paper due to the fact that it is by far the most popular programming language for bioinformatics[13].

2 Methods

2.1 Hardware and Software

The tests were run on a laptop with the following specifications:

- CPU Model: Intel Core i5-10210U
- CPU Cores: 4[14]
- CPU Threads: 8[14]
- CPU Base Frequency: 1.60 GHz[14]
- CPU Max Turbo Frequency: 4.20 GHz[14]
- RAM: 16 GB DDR4 2933 MT/s
- Storage: 512 GB NVMe SSD
- Operating System: Windows 11 Pro 64-bit
- IDE: Visual Studio Code (version 1.109.5)

In order to ensure minimal interference, the test were run with no other applications (other than the IDE itself) running in the background, and with the laptop plugged in and heavy background processes like Windows Update disabled.

2.2 Test Data

To ensure a comprehensive evaluation of the algorithms, synthetic test data was generated using three different alphabets and five different scenarios, each designed to cover a wide range of cases that could be encountered in real-world applications, with a particular focus on the field of bioinformatics, where the LCS problem is particularly relevant due to the nature of DNA and RNA sequences being represented as strings.

The alphabets used for the test data were as follows:

Alphabet Name	**Description**	**Characters**
Full	Every lowercase letter	a, b, c, ..., z
DNA	The four DNA bases	A, C, G, T
Disjoint	Two disjoint sets of the lowercase letters	String 1: a, b, c, ..., m String 2: n, o, p, ..., z

Table 2.1: Description of the alphabets used for generating test data

Five separate scenarios were tested, in order to cover as wide a range of cases as possible:

Scenario Name	**Description**	**Alphabet**
random_uniform	Both strings are generated randomly with a uniform distribution of characters	Full
mutated_implant	A common motif is implanted in both strings with mutations	DNA
repetitive_with_noise	Strings are formed by repeating a pattern with some mutations	DNA
near_identical	Strings are nearly identical, with minor mutations preferentially at the edges	Full
disjoint_alphabet	Strings are generated from completely disjoint alphabets	Disjoint

Table 2.2: Description of test scenarios used for benchmarking

The justifications for the choice of these scenarios are:

- The *random_uniform* scenario serves as a baseline for the performance of the algorithms on completely random data.
- The *mutated_implant* scenario simulates a common real-world case in bioinformatics, where two otherwise unrelated long sequences share a relatively short common motif between them at unrelated positions[15,16]. It's basically the "needle in a haystack" scenario.
- The *repetitive_with_noise* scenario simulates another common case in bioinformatics, where sequences are formed by repeating a pattern (tandem repeats[17]) with some mutations, which can lead to multiple short LCSs.

- The *near_ identical* scenario tests the performance of the algorithms on strings that are nearly identical, so that there would be one very long LCS which would have to be held in memory.

- The *disjoint_ alphabet* scenario tests the case where the two strings have no common characters, which should result in an LCS of length 0. The algorithms would have to traverse the entire data structure to determine this. It also serves as a sanity check to ensure that the algorithms are correctly identifying the lack of common substrings.

2.3 Design

The algorithms themselves were implemented in pure Python (version 3.13.12), without the aid of any external libraries (although external libraries were used for gathering the results and plotting the graphs).

The SAM was implemented using the standard on-line construction algorithm[18]. To find the LCS of the first string and the second string, the SAM was created with the first string, and then traversed through by the second string, keeping track of the longest match found.

To give the Suffix Array (SA) the best chance possible, the SA was constructed using the Suffix Array with Induced Sorting (SA-IS) algorithm, the current state-of-the-art algorithm for constructing it in linear time[9]. The SA was constructed for the concatenated string of the two strings being compared, with a sentinel character in between them (\0 in this case) to separate them. The LCP Array was constructed using Kasai's algorithm[10], which is a linear time algorithm for constructing it from the SA. The ESA was essentially implemented as a combination of the SA and the LCP Array.

25 string pairs each of lengths 100, 500, 1000, 5000, and 10,000 characters were generated for each scenario, and each string pair was tested on both algorithms **20 times** to try and ensure statistical significance of the results. To ensure reproducibility, the random seed used for generating the string pairs was fixed at 42 for all scenarios and string lengths.

The testing code was designed to run both algorithms on the same string pairs, and to measure both the time taken and the peak memory usage for each run. The time taken was measured using the `time` module, while the peak memory usage was measured using the `tracemalloc` module, which is a built-in Python module for tracking memory allocations. The index size for each algorithm was computed using the `sys.getsizeof` function, which returns the size of an object by recursively summing the sizes of all referenced objects.

The full testing code, including a list of required packages and instructions for running it, is provided in the supplementary materials. A complete list of generated string pairs with their corresponding LCS values, as well as the raw test results, are also included. Additionally, the code used for plotting the results (using the `matplotlib` library) is provided.

3 Results

3.1 Choice of Central Tendency

The data collected seems to be heavily right-skewed, with some outliers. This is irrespective of the scenario, string length, algorithm, or even the type of measurement (build time, query time, total time, or memory usage).

This is likely due to cache misses, other background processes running on the computer, or other random factors that can affect the performance of the algorithms in a non-deterministic way. Therefore, the median was chosen as the measure of central tendency for the results, as the data is not normally distributed and the median is more robust to outliers than the mean. And since the median is being used, the interquartile range (IQR) was chosen as the measure of variability.

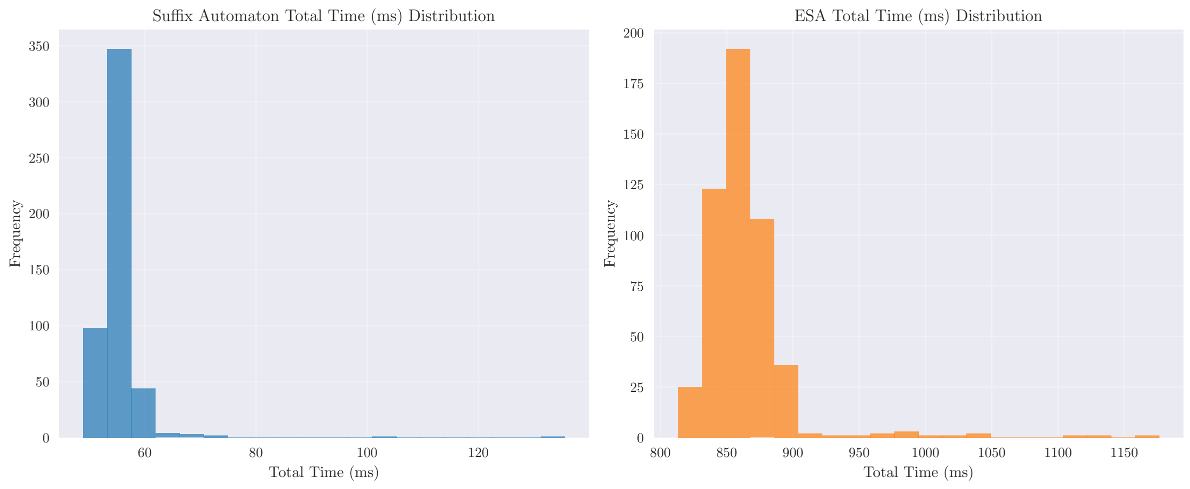

Figure 3.1: Distribution of total time for the SAM and the ESA for the random uniform scenario and string length 10000, as an example of the right-skewed nature of the data

3.2 Time

The time taken to construct the SAM and the ESA and to find the LCS was measured for each scenario and string length, and the results are shown in the following plots.

3.2.1 Build Time

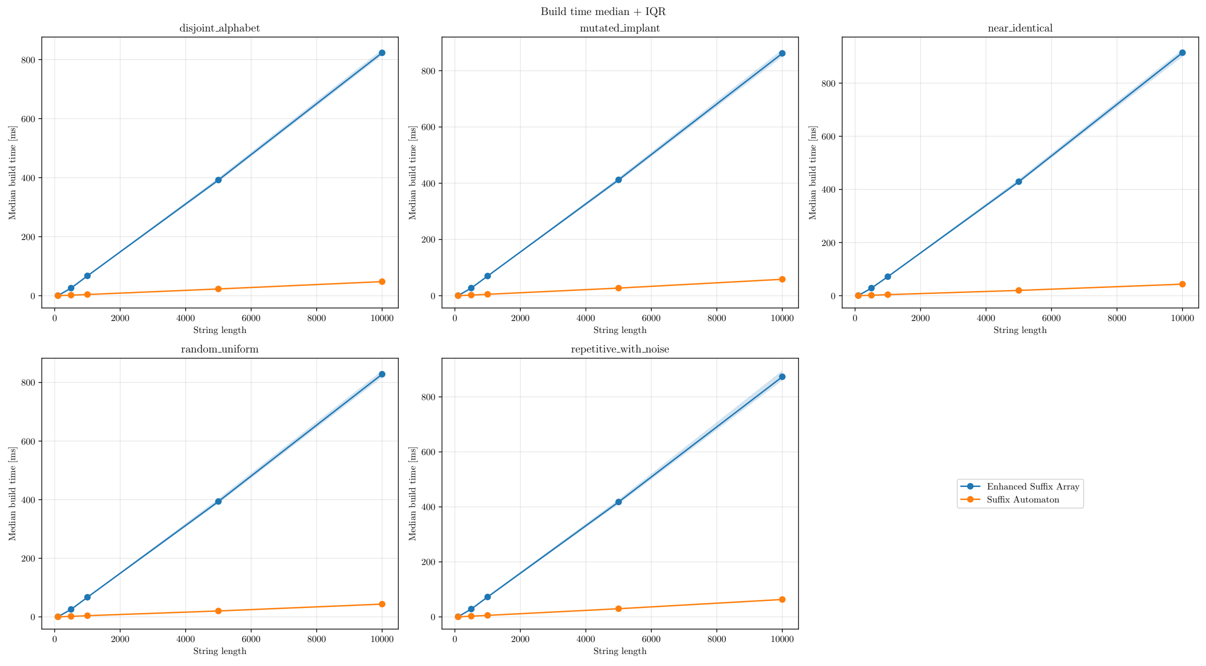

Figure 3.2: Median and interquartile range of build time for the SAM and the ESA across different scenarios and string lengths

3.2.2 Query Time

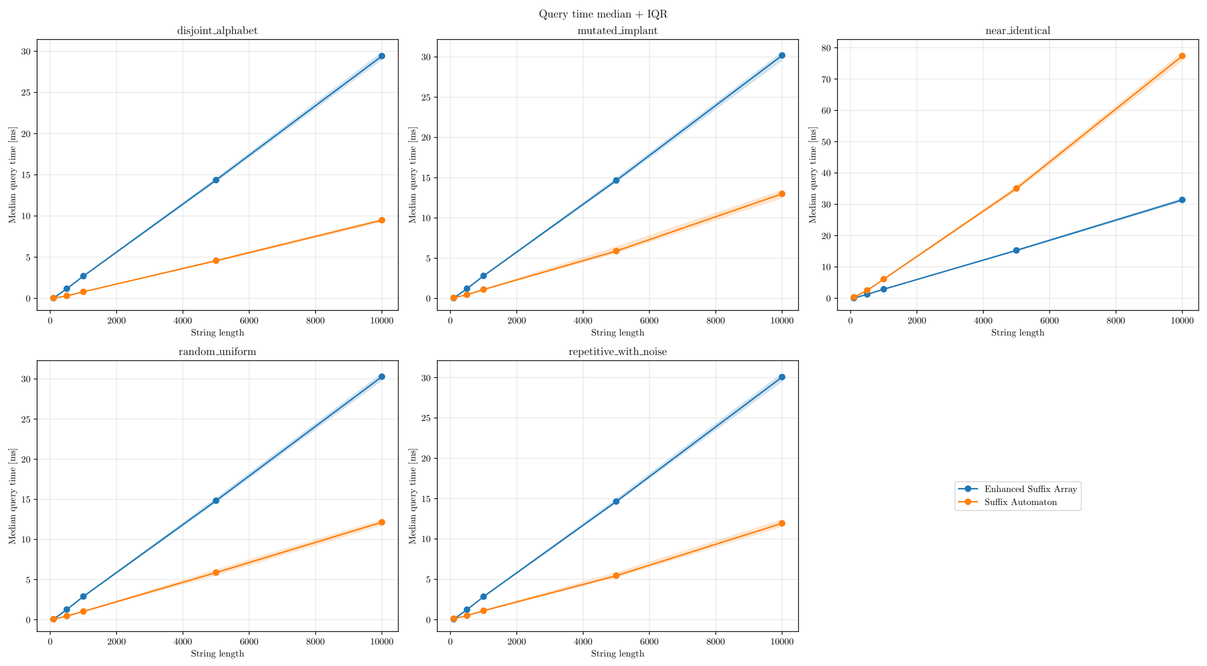

Figure 3.3: Median and interquartile range of query time for the SAM and the ESA across different scenarios and string lengths

3.2.3 Total Time

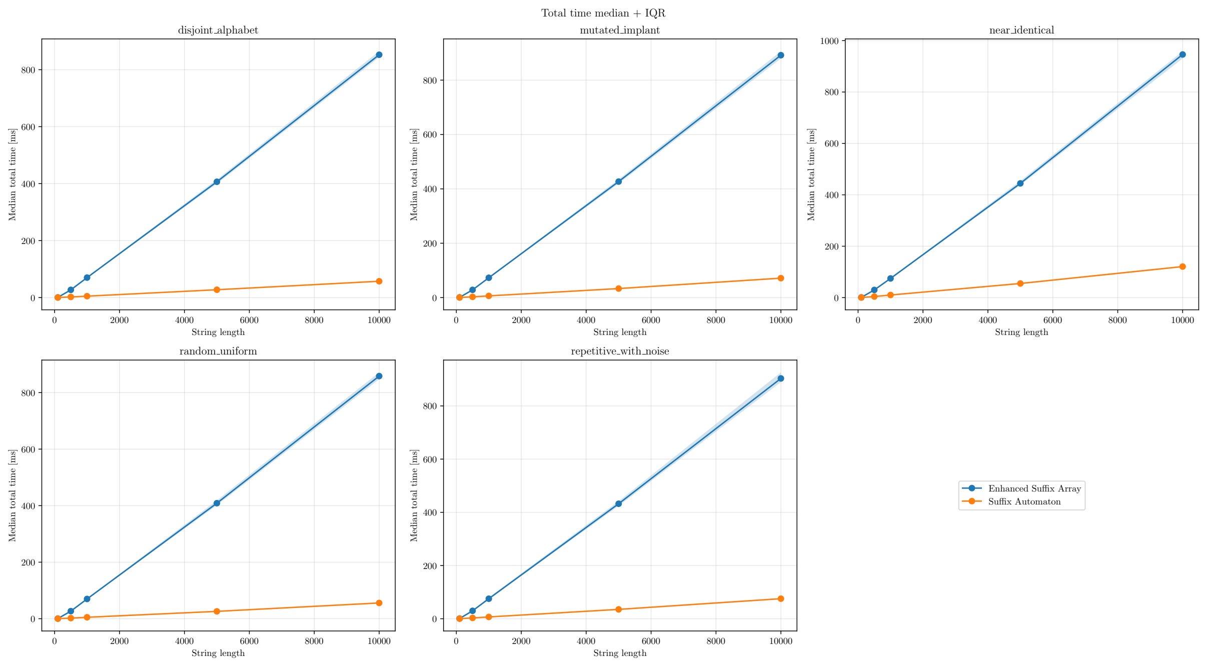

Figure 3.4: Median and interquartile range of total time for the SAM and the ESA across different scenarios and string lengths

Bibliography

[1] David J. Lipman and William R. Pearson. „Rapid and Sensitive Protein Similarity Searches“. In: *Science* 227.4693 (1985), pp. 1435–1441. DOI: `10.1126/science.2983426`. eprint: `https://www.science.org/doi/pdf/10.1126/science.2983426`. URL: `https://www.science.org/doi/abs/10.1126/science.2983426`.

[2] Amihood Amir et al. „Dynamic and internal longest common substring“. In: *Algorithmica* 82.12 (2020), pp. 3707–3743.

[3] Peter Weiner. „Linear pattern matching algorithms“. In: *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. 1973, pp. 1–11. DOI: `10.1109/SWAT.1973.13`.

[4] Edward M. McCreight. „A Space-Economical Suffix Tree Construction Algorithm“. In: *J. ACM* 23.2 (Apr. 1976), pp. 262–272. ISSN: 0004-5411. DOI: `10.1145/321941.321946`. URL: `https://doi.org/10.1145/321941.321946`.

[5] Mehryar Mohri, Pedro Moreno, and Eugene Weinstein. „General suffix automaton construction algorithm and space bounds“. In: *Theoretical Computer Science* 410.37 (2009). Implementation and Application of Automata (CIAA 2007), pp. 3553–3562. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2009.03.034`. URL: `https://www.sciencedirect.com/science/article/pii/S0304397509002370`.

[6] A. Blumer et al. „The smallest automation recognizing the subwords of a text“. In: *Theoretical Computer Science* 40 (1985). Eleventh International Colloquium on Automata, Languages and Programming, pp. 31–55. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(85)90157-4`. URL: `https://www.sciencedirect.com/science/article/pii/0304397585901574`.

[7] Maxime Crochemore and Wojciech Rytter. *Text algorithms*. Maxime Crochemore, 1994.

[8] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. „Replacing suffix trees with enhanced suffix arrays“. In: *Journal of Discrete Algorithms* 2.1 (2004). The 9th International Symposium on String Processing and Information Retrieval, pp. 53–86. ISSN: 1570-8667. DOI: `https://doi.org/10.1016/S1570-8667(03)00065-0`. URL: `https://www.sciencedirect.com/science/article/pii/S1570866703000650`.

[9] Ge Nong, Sen Zhang, and Wai Hong Chan. „Two Efficient Algorithms for Linear Time Suffix Array Construction“. In: *IEEE Transactions on Computers* 60.10 (2011), pp. 1471–1484. DOI: 10.1109/TC.2010.188.

[10] Toru Kasai et al. „Linear-time longest-common-prefix computation in suffix arrays and its applications“. English. In: *Combinatorial Pattern Matching - 12th Annual Symposium, CPM 2001, Proceedings*. Ed. by Amihood Amir et al. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Publisher Copyright: © Springer-Verlag Berlin Heidelberg 2001.; 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001 ; Conference date: 01-07-2001 Through 04-07-2001. Springer Verlag, 2001, pp. 181–192. ISBN: 3540422714. DOI: 10.1007/3-540-48194-x_17.

[11] Udi Manber and Gene Myers. „Suffix arrays: a new method for on-line string searches“. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '90. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327. ISBN: 0898712513.

[12] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[13] Diego Mariano et al. „A Brief History of Bioinformatics Told by Data Visualization“. In: *Advances in Bioinformatics and Computational Biology*. Ed. by João C. Setubal and Waldeyr Mendes Silva. Cham: Springer International Publishing, 2020, pp. 235–246. ISBN: 978-3-030-65775-8. URL: https://bioinfo.dcc.ufmg.br/history/.

[14] Intel Corporation. *Intel® Core™ i5-10210U Processor (6M Cache, up to 4.20 GHz) Specifications*. 2019. URL: https://www.intel.com/content/www/us/en/products/sku/195436/intel-core-i510210u-processor-6m-cache-up-to-4-20-ghz/specifications.html (visited on 02/26/2026).

[15] Jerzy W. Jurka. „Chapter Thirty-Nine - Approaches to Identification and Analysis of Interspersed Repetitive DNA Sequences“. In: *Automated DNA Sequencing and Analysis*. Ed. by Mark D. Adams, Chris Fields, and J. Craig Venter. San Diego: Academic Press, 1994, pp. 294–298. ISBN: 978-0-08-092639-1. DOI: https://doi.org/10.1016/B978-0-08-092639-1.50043-5. URL: https://www.sciencedirect.com/science/article/pii/B9780080926391500435.

[16] Barbara McClintock. „The origin and behavior of mutable loci in maize“. In: *Proceedings of the National Academy of Sciences* 36.6 (1950), pp. 344–355. DOI: 10.1073/pnas.36.6.344. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.36.6.344. URL: https://www.pnas.org/doi/abs/10.1073/pnas.36.6.344.

[17] Jorge Duitama et al. „Large-scale analysis of tandem repeat variability in the human genome“. In: *Nucleic Acids Research* 42.9 (Mar. 2014), pp. 5728–5741. ISSN: 0305-1048. DOI: `10.1093/nar/gku212`. eprint: `https://academic.oup.com/nar/article-pdf/42/9/5728/25346735/gku212.pdf`. URL: `https://doi.org/10.1093/nar/gku212`.

[18] A. Blumer et al. „Building the minimal DFA for the set of all subwords of a word on-line in linear time“. In: *Automata, Languages and Programming*. Ed. by Jan Paredaens. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 109–118. ISBN: 978-3-540-38886-9.

AI Declaration

The usage of AI tools within this project is documented here. I solemnly declare that I have documented all interactions with AI tools, including the prompts used and the outputs received.

System	Prompt	Usage
GitHub Copilot 1	Asked to delete the existing project and provide a simple LaTeX template with a shared preamble, a Chapters subdirectory, TOC, glossary, abbreviations, and Roman numerals for non-content pages.	Template structure and LaTeX setup provided
GitHub Copilot 2	Requested additional title page lines (WAB header, reviewer, module) for the main document.	Title page metadata and layout updated
GitHub Copilot 3	Requested uppercase Roman numerals for front matter and lowercase Roman numerals for back matter page numbering.	Page numbering adjusted in main and exposee
GitHub Copilot 4	Requested exposee title page to include shared info fields (WAB header, department, module, reviewer).	Exposee title page updated to include shared metadata
GitHub Copilot 5	Requested adding the provadis-hochschule.pdf logo to the top right corner of both main and exposee title pages.	Logo added to top right of both title pages
GitHub Copilot 6	Reported that Glossary and Abbreviations sections were missing from compiled output.	Added example `\newacronym` entries to abbreviations.tex
GitHub Copilot 7	Reported that glossary section still not appearing in final PDF.	Updated settings.tex to include `automake` option in glossaries package to enable automatic glossary generation

System	Prompt	Usage
GitHub Copilot 8	Requested Python benchmark code for LCS comparison with multiple synthetic scenarios (random strings, implanted mutated substrings, etc.)	Created benchmark script and documentation in Code folder
GitHub Copilot 9	Asked to add a progress-bar-like output.	Added optional single-line in-place progress output with percentage, completed/total steps, elapsed time, ETA, and `-progress`/`-no-progress` flags
GitHub Copilot 10	Reported that ETA was inaccurate due to varying string lengths	Reworked ETA estimation to be algorithm- and length-aware using observed per-bucket runtimes
GitHub Copilot 11	Requested separate build and query timing (and plots).	Refactored benchmark to record build/query time separately and added total time summaries
GitHub Copilot 12	Requested plots using statistics beyond the mean (median, standard deviation, etc.).	Expanded summary stats and generated mean+std and median+IQR plots alongside memory plots
GitHub Copilot 13	Requested splitting benchmarking and plotting so plot generation can run separately, with a shared CLI entry point and default run-then-plot behavior.	Added CLI mode switch (run/plot/both) to reuse saved CSV results for plotting without rerunning benchmarks
GitHub Copilot 14	Requested an error if the two algorithms return different LCS result strings for the same input.	Added strict substring equality check (in addition to length) and raise an error on mismatch
GitHub Copilot 15	Requested that, in case of multiple valid LCS results, both algorithms return all results and compare the sets.	Updated both algorithms to return all max-length substrings and compare result sets for correctness
GitHub Copilot 16	Requested refactoring the benchmark script into helper modules under a helpers subdirectory, with separate algorithm imports and plotting helper, and clean top-level imports in the main file.	Split benchmark code into dedicated helper modules (SAM, ESA, plotting, generation, statistics, benchmarking) and simplified main script orchestration
GitHub Copilot 17	Requested mean, median, standard deviation, and IQR for memory usage plots as well.	Added memory quartile aggregation and generated memory mean+SD plus median+IQR plots
GitHub Copilot 18	Asked for a practical memory/space-complexity metric and whether build-time and query-time memory can be measured separately.	Added separate build/query phase memory metrics and a persistent index-size metric, with aggregation and plots for comparison

System	Prompt	Usage
GitHub Copilot 19	Asked whether adding total time (build + query) columns and graphs with full statistics would make sense.	Added full total-time statistics (mean, median, std, IQR, Q1, Q3) to summaries and created dedicated total-time plots
GitHub Copilot 20	Requested moving plot legends away from the top center because they obscured titles.	Repositioned all figure legends to the right side outside the plotting area to keep titles unobstructed
GitHub Copilot 21	Requested an additional output file containing both generated strings and their LCS value(s).	Added export of one CSV per generated case with ‘s‘, ‘t‘, LCS length, and all LCS substrings
GitHub Copilot 22	Asked to replace the lambda with a tuple-key-based approach.	Replaced lambda-based sorting with a tuple-key-based approach for suffix array construction, improving performance
GitHub Copilot 23	Asked to refactor the plotting code to reduce repetition and make it more modular.	Refactored plotting code to use a single generic plotting function
GitHub Copilot 24	Asked to replace the suffix array construction with the SA-IS algorithm.	Replaced doubling-based suffix array construction with an SA-IS implementation (linear-time induced sorting with LMS recursion), keeping the same public API

Declaration of Authorship

I hereby confirm that I have personally and independently prepared the present work and have not used any sources or aids other than those specified. All passages taken verbatim or in substance from other sources are identified as such. The drawings, illustrations and tables in this work are created by me or have been provided with an appropriate source reference. This work has not been submitted by me to any other university in the same or similar form for the acquisition of an academic degree.

Frankfurt, February 26, 2026 ....

Rubin Chempananickal James