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 28, 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, 5

RNA Ribonucleic Acid, a molecule that is transcribed from DNA and which then gets translated into proteins. 1, 5

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, 7, 16

Abbreviations

DAWG Directed Acyclic Word Graph.

ESA Enhanced Suffix Array.

IQR Interquartile Range.

LCP Longest Common Prefix.

LCS Longest Common Substring.

SA Suffix Array.

SA-IS Suffix Array with Induced Sorting.

SAM Suffix Automaton.

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.

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 concrete 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 in the field of bioinformatics[13].

2 Methods

2.1 Hardware Specifications

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

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

2.2 Python Environment

Here are the details of the Python environment used for the implementation and testing of the algorithms:

Python Version	3.13.12
Operating System	Windows 11 Pro 64-bit (version 25H2)
IDE	Visual Studio Code (version 1.109.5)
Environment Manager	Miniforge (conda version 26.1.0)
Packages Used	**Package** / **Version**: time built-in; tracemalloc built-in; sys built-in; matplotlib 3.10.8; pandas 3.0.1; jupyter 1.1.1; Jinja2 3.1.6

Package	Version
time	built-in
tracemalloc	built-in
sys	built-in
matplotlib	3.10.8
pandas	3.0.1
jupyter	1.1.1
Jinja2	3.1.6

Table 2.1: Details of the Python environment used for implementation and testing

2.3 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 2.2: 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 2.3: 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.

2.4 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 requirements file 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.

Tables with the exact values of the median and IQR for all measurements are included in the appendix (page iv) grouped by scenario, and the plots in the main text are based on these values. An even more detailed summary is available as a CSV file in the supplementary materials, which includes other statistics such as the mean and standard deviation.

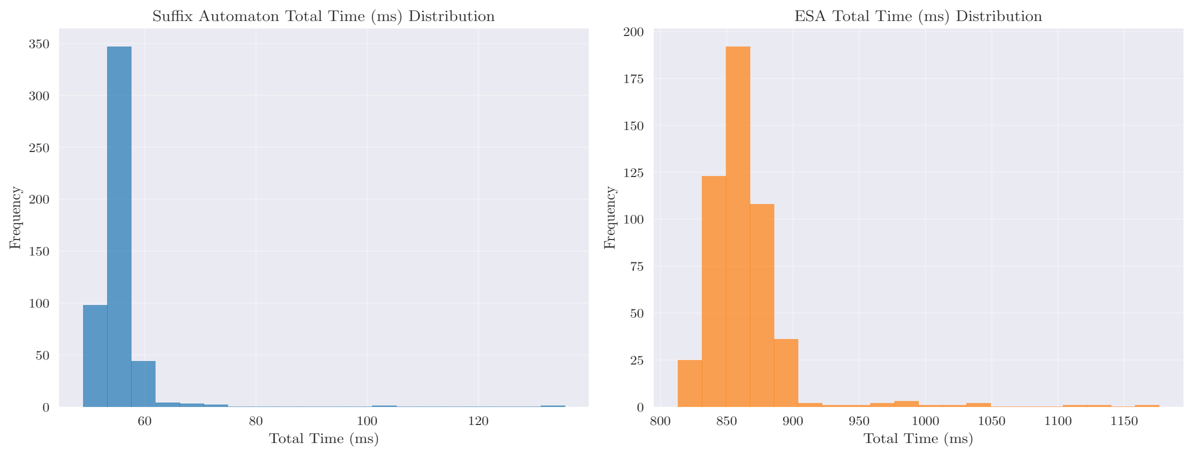

Figure 3.1: Distribution of total time for the SAM and the ESA for the random uniform scenario and string length 10000, as an example of the right-skewed nature of the data

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.

The build time, query time, and total time are shown separately to give a more detailed view of the performance of the algorithms. The median is shown as a line, and the IQR is shown as a shaded area around the line. Because the IQR is very small for most of the data points, it may not be visible in the plots.

3.2.1 Build Time

The build time is the time taken to construct the data structure (the SAM or the ESA) from the string(s).

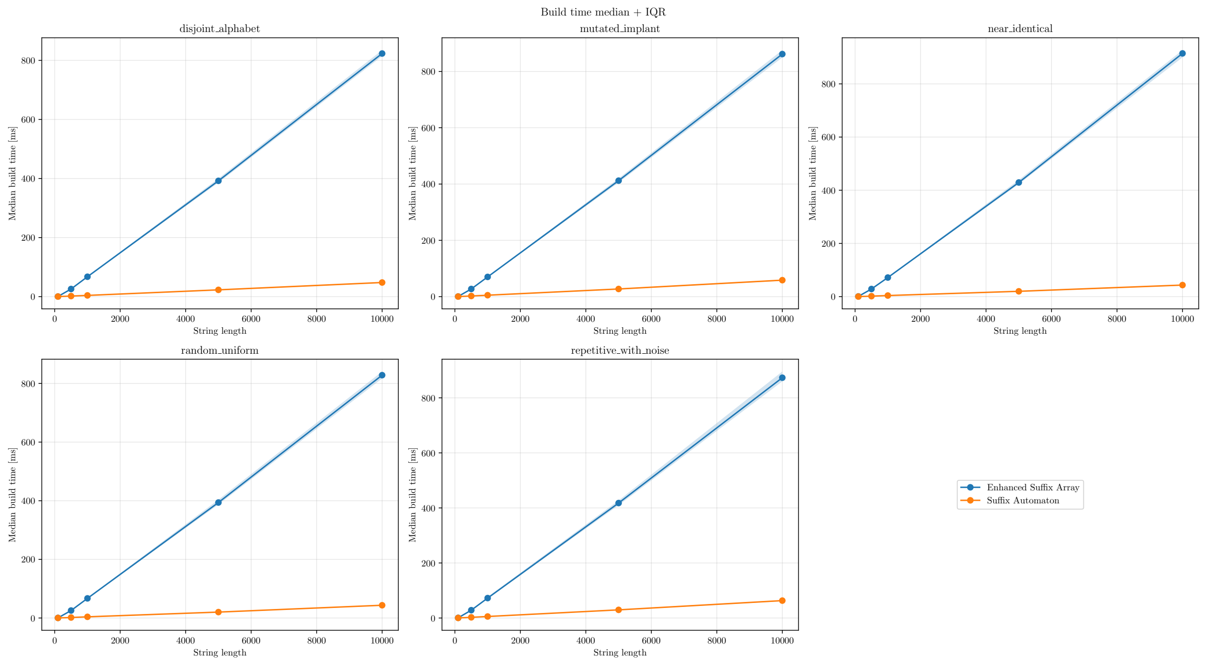

Figure 3.2: Median and interquartile range (IQR) of build time for the SAM and the ESA across different scenarios and string lengths

As can be seen from the plot, the build time for both algorithms tend to follow a linear trend with respect to the string length (as is to be expected), but the ESA has a much higher build time than the SAM, especially as the string length increases.

3.2.2 Query Time

The query time is the time taken to find the LCS of the two strings using the constructed data structure.

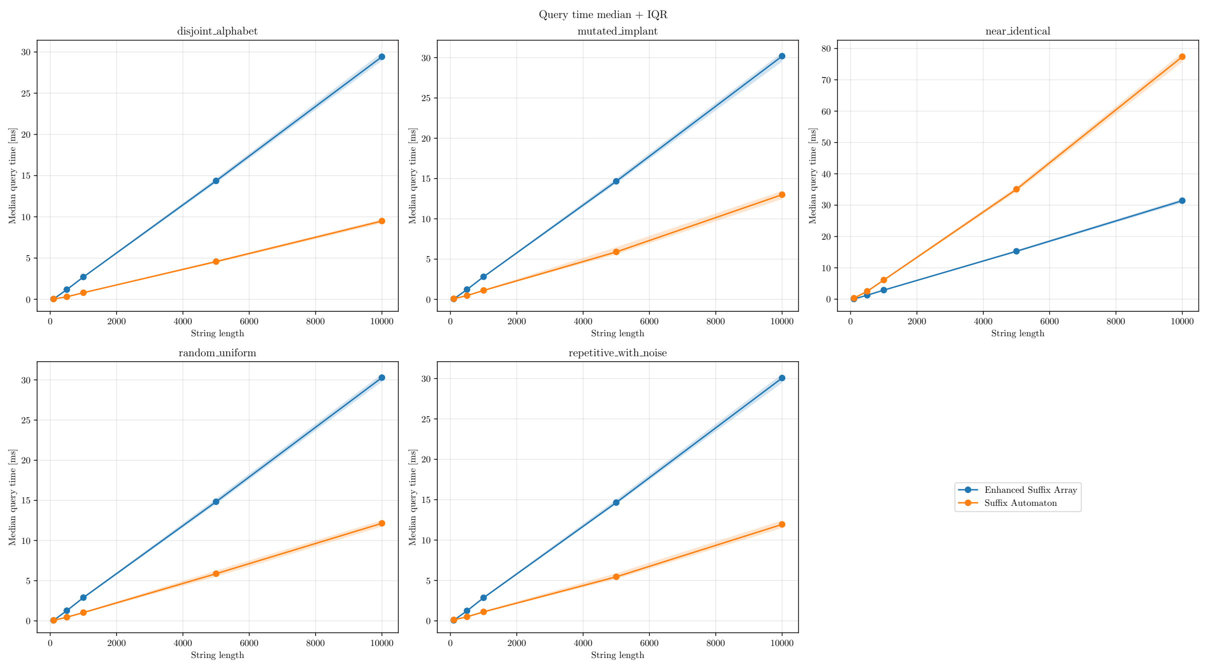

Figure 3.3: Median and interquartile range (IQR) of query time for the SAM and the ESA across different scenarios and string lengths

The query time for both algorithms also tends to follow a linear trend with respect to the string length, and for most cases, the SAM has a shorter query time than the ESA, with near_identical being the only exception. This outlier can be explained by the fact that in the near_identical scenario, where the LCS is singular and very long, causing the SAM to have to traverse a long path in the automaton, whereas the ESA can find the LCS by directly inspecting the LCP between the relevant suffixes, and that too, only once. This leads to the interesting observation that *the query time of the SAM grows in proportion to the length of the LCS*, at least in Python.

Also note that the query time for both algorithms is extremely small compared to the build time, and even with string pairs that are each 10,000 characters long, the median query time remained under 0.1 seconds.

3.2.3 Total Time

The total time represents the time from the start of the construction of the data structure to the end of finding the LCS, so it is essentially the sum of the build time and the query time.

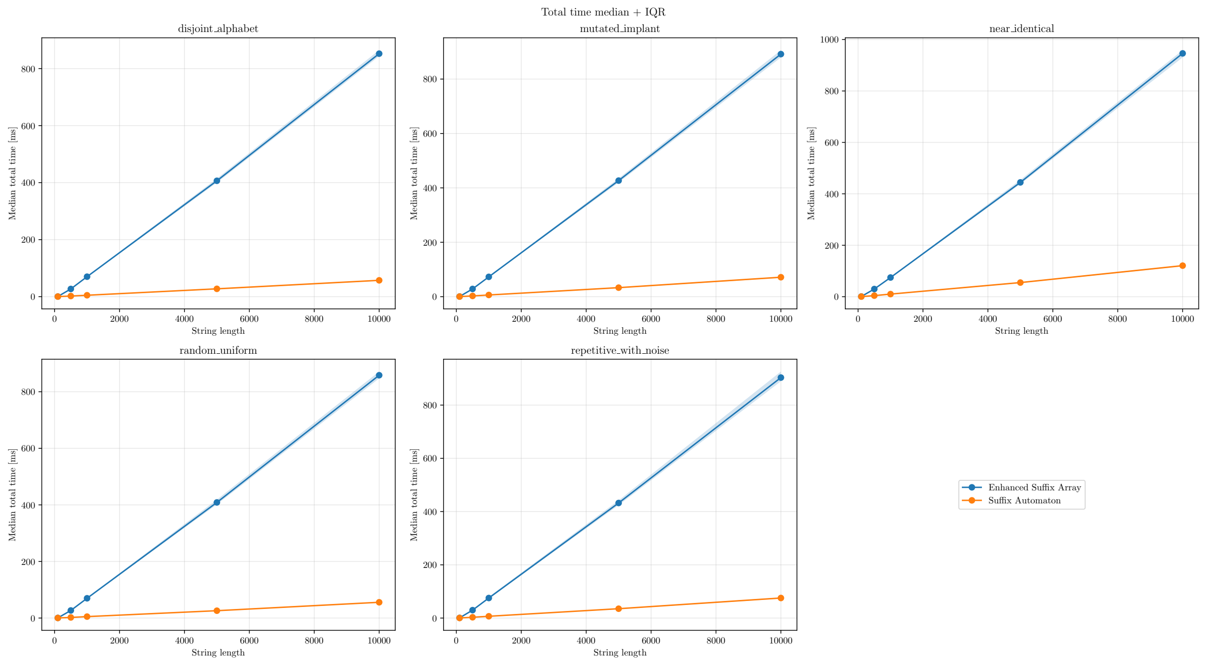

Figure 3.4: Median and interquartile range (IQR) of total time for the SAM and the ESA across different scenarios and string lengths

Because the build time was so high for the ESA compared to the SAM, the total time for the ESA is also much higher than that of the SAM across all scenarios and string lengths, with the difference becoming more pronounced as the string length increases.

This is in spite of the fact that the ESA was built with the SA-IS and Kasai's algorithms, avoiding the traditional naive algorithms which would have used Python's built-in sorting, rendering the time complexity to be $O(nlog\ n)$ for a combined string of length n.

3.3 Memory Usage

The memory usage was measured for both the build phase and the query phase, as well as the total memory usage, which is the peak memory usage during the entire process. The index size was also measured, which is the size of the data structure (the SAM or the ESA) in memory.

3.3.1 Index Size

The index size is measured using the `sys.getsizeof` function, which returns the size of an object in bytes, including the size of all referenced objects recursively.

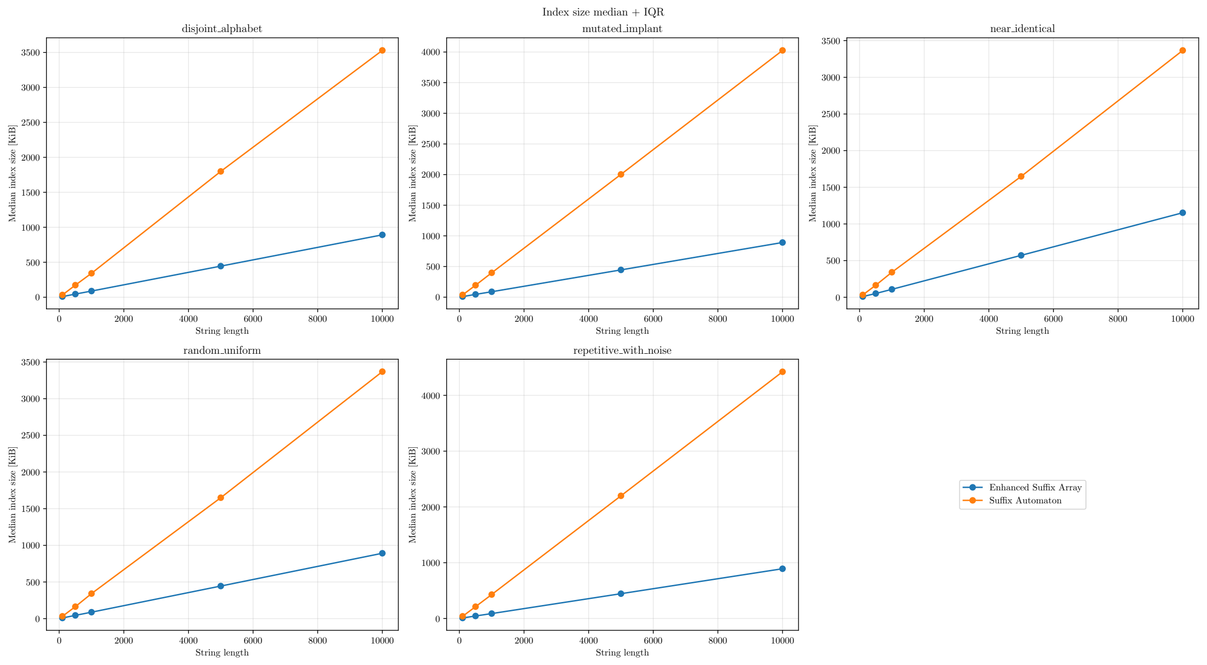

Figure 3.5: Median and interquartile range (IQR) of index size for the SAM and the ESA across different scenarios and string lengths

As can be seen from the plot, the index size for the SAM is much larger than that of the ESA. This is to be expected, as the SAM is a much more complex data structure, being a deterministic finite automaton, while the ESA is essentially just two arrays of integers.

3.3.2 Peak Memory Usage

The peak memory usage is measured using the `tracemalloc` module, which tracks memory allocations in Python.

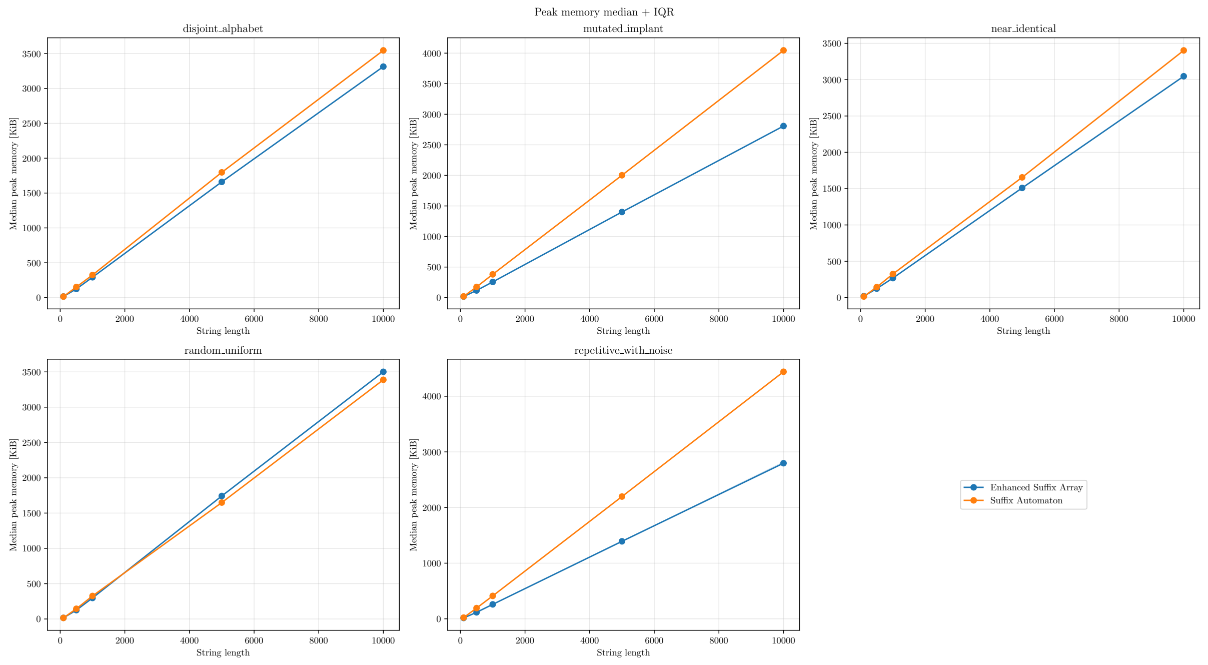

Figure 3.6: Median and interquartile range (IQR) of peak memory usage for the SAM and the ESA across different scenarios and string lengths

This, on the other hand, was a much more surprising result, because as can be seen from the plot, the peak memory usage was comparable between the two algorithms across three out of the five tested scenarios, with the SAM even having a slightly lower peak memory usage than the ESA in the case of the random_uniform scenario.

And of the two scenarios where the SAM had a higher peak memory usage than the ESA, the difference was not as pronounced as the difference in index size, thereby lending credence to the argument that a pure Python implementation of the ESA does not yield a significant advantage in terms of memory usage in actally memory-constrained environments, where the peak memory usage is what will determine whether the algorithm can run or not, rather than the final index size.

4 Discussion and Conclusion

4.1 Results and Discussion

In this paper, two algorithms for solving the longest common substring (LCS) problem **between two strings** were compared: the Suffix Automaton (SAM) and the Enhanced Suffix Array (ESA). The algorithms were implemented in pure Python and tested on a variety of scenarios and string lengths, with the results measured in terms of build time, query time, total time, peak memory usage, and index size.

The results showed that the SAM generally outperformed the ESA in terms of build time and total time, while the query time was more variable depending on the scenario. The SAM had a shorter query time than the ESA in most cases, except for the near_identical scenario where the ESA had a shorter query time due to the nature of the LCS being singular and very long.

In terms of memory usage, while SAM had a much larger index size than the ESA, with the SAM reaching above 4 MiB median for strings of length 10,000, (compared to the ESA which stayed barely above 1 MiB median for the same string length), the peak memory usage of the SAM and the ESA were more comparable in most scenarios, negating any potential benefit of the ESA in memory-constrained environments.

The SAM's performance advantage in build time can be attributed to its more straightforward construction algorithm, which processes the string character by character while maintaining a relatively simple state structure. In contrast, the ESA construction involves first building the suffix array using the SA-IS algorithm, followed by computing the LCP array using Kasai's algorithm, introducing additional computational overhead despite both algorithms being theoretically linear.

The memory results revealed an interesting discrepancy between theoretical expectations and practical outcomes. Despite the ESA having a significantly smaller index size (barely above 1 MiB for 10,000-character strings compared to the SAM's 4+ MiB), the peak memory usage during construction and query remained comparable between the two approaches. This can be attributed to Python's memory management system and its substantial per-object overhead. As a dynamically typed, interpreted language, Python allocates memory for objects with considerable metadata, and the temporary data structures created during algorithm execution consume memory that is not reflected in the final index size measurements. This Python-specific behavior suggests that the ESA's theoretical space advantage may be diminished in practice when using high-level languages with significant runtime overhead.

Another noteworthy observation was that the query time of the SAM appeared to grow proportionally to the length of the resulting LCS. This behavior, particularly evident in the near_identical scenario where the ESA outperformed the SAM, suggests that the SAM's query performance is influenced not only by the input string length but also by the characteristics of the match itself. This could be due to the way the Python implementation constructs the result set of matching substrings, and warrants further investigation to determine whether this pattern persists in lower-level language implementations or with different result collection strategies.

4.2 Limitations

It is important to note that this study focused specifically on the LCS problem **between exactly two strings** in a Python implementation. The results and conclusions are therefore most applicable to this particular use case and programming environment.

The choice of Python as the implementation language, while justified by its prevalence in bioinformatics, introduces language-specific characteristics that may not generalize to implementations in lower-level languages such as C[19] or Rust[20], where memory management is more explicit and overhead is significantly reduced. Additionally, the benchmark scenarios, while diverse, represent a finite set of test cases with a focus on bioinformatics-relevant patterns, and may not capture the full spectrum of possible input characteristics that could influence algorithm performance.

Furthermore, this comparison examined relatively straightforward implementations of both algorithms without exploring potential optimizations such as specialized memory allocators.

4.3 Practical Considerations

While the benchmarks demonstrated the SAM's overall advantage for the specific two-string LCS problem, the choice between these algorithms in practice should be guided by the particular use case at hand.

If multiple strings need to be compared against the same "reference" string, the SAM becomes significantly more attractive. The automaton needs to be constructed only once for the reference string and can then be reused for all subsequent comparisons, making the construction time effectively "amortized" across all queries. This makes the SAM particularly well-suited for applications such as pattern matching in genomic databases, where a single reference genome is queried repeatedly against many sample sequences.

Conversely, if the LCS of more than two strings needs to be found simultaneously, the ESA still holds a distinct advantage. The ESA can be constructed for an arbitrary number of strings by concatenating them with unique sentinel characters, so long as one can find enough sentinels that are lexicographically smaller than any character in the input strings and not present in the strings themselves. The equivalent functionality for the SAM, the "Generalized Suffix Automaton"[5] - is a substantially more complex data structure that maintains the same $O(|\sum_{i=1}^{k} n_i|)$ construction complexity as the ESA, where n_i is the length of string i and k is the number of strings. Given the increased implementation complexity without a corresponding performance benefit, the ESA would likely be preferable for multi-string LCS problems, unless a "sentinel-free" approach is specifically needed.

4.4 Future Work

Due to the narrow scope of this paper, more advanced algorithms like the FM-Index[21] were not included in the comparisons, but it could be interesting to compare against in future work, especially given that it is built on top of the ESA idea and optimized for using minimal space.

Additionally, a comparison between these algorithms when they are written in Cython[22] would be worth further exploration, as many performance critical Python packages are implemented in Cython to achieve a balance between Python's ease of use and C's efficiency.

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.

[19] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[20] Nicholas D. Matsakis and Felix S. Klock. „The rust language“. In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: `10.1145/2692956.2663188`. URL: `https://doi.org/10.1145/2692956.2663188`.

[21] P. Ferragina and G. Manzini. „Opportunistic data structures with applications“. In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 2000, pp. 390–398. DOI: `10.1109/SFCS.2000.892127`.

[22] Stefan Behnel et al. „Cython: The Best of Both Worlds“. In: *Computing in Science and Engg.* 13.2 (Mar. 2011), 31–39. ISSN: 1521-9615. DOI: `10.1109/MCSE.2010.118`. URL: `https://doi.org/10.1109/MCSE.2010.118`.

Appendix

The mean, standard deviation, median, and interquartile range (IQR) for each scenario and for the two tested algorithms, are presented in the following tables.

For further results and the raw data itself, please refer to the supplementary materials.

Disjoint Alphabet

Build Time Comparison
Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Build Time Mean (ms)	Build Time SD (ms)	Build Time Median (ms)	Build Time IQR (ms)
ESA	100	0.5106	0.1265	0.4583	0.1391
SAM	100	0.2354	0.0501	0.2236	0.0222
ESA	500	26.2949	3.0448	25.8245	1.1181
SAM	500	1.9801	0.3657	1.8739	0.2254
ESA	1000	67.9895	4.1817	67.3458	3.2995
SAM	1000	4.1412	0.2445	4.1075	0.2836
ESA	5000	394.3566	15.9704	392.0425	14.0880
SAM	5000	23.0649	3.3611	22.9133	1.4102
ESA	10000	827.6511	29.0246	823.1218	20.5804
SAM	10000	47.9052	4.0663	47.7820	2.2490

Query Time Comparison
Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Query Time Mean (ms)	Query Time SD (ms)	Query Time Median (ms)	Query Time IQR (ms)
ESA	100	0.0338	0.0076	0.0319	0.0023
SAM	100	0.0331	0.0096	0.0312	0.0029
ESA	500	1.2148	0.1354	1.1702	0.0906
SAM	500	0.3170	0.1573	0.2997	0.0154
ESA	1000	2.7226	0.1462	2.7022	0.1424
SAM	1000	0.7997	0.0366	0.7935	0.0381
ESA	5000	14.4301	0.6275	14.3621	0.5594
SAM	5000	4.6049	0.6684	4.5716	0.2821
ESA	10000	29.5849	1.2168	29.4213	0.9918
SAM	10000	9.5072	0.7182	9.4911	0.4564

Total Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Total Time Mean (ms)	Total Time SD (ms)	Total Time Median (ms)	Total Time IQR (ms)
ESA	100	0.5444	0.1309	0.4907	0.1391
SAM	100	0.2684	0.0566	0.2555	0.0253
ESA	500	27.5098	3.1136	27.0282	1.1357
SAM	500	2.2971	0.4194	2.1760	0.2493
ESA	1000	70.7121	4.2814	70.0603	3.5061
SAM	1000	4.9409	0.2572	4.9126	0.3063
ESA	5000	408.7867	16.3983	406.4943	14.3276
SAM	5000	27.6698	3.5782	27.4650	1.7086
ESA	10000	857.2359	29.6781	852.5146	21.4592
SAM	10000	57.4125	4.6418	57.2604	2.6455

Build Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Build Peak Memory Mean (KiB)	Build Peak Memory SD (KiB)	Build Peak Memory Median (KiB)	Build Peak Memory IQR (KiB)
ESA	100	16.4984	1.3383	16.0645	0.4219
SAM	100	14.7741	0.3781	14.8047	0.5469
ESA	500	124.0991	0.9039	124.1738	0.6562
SAM	500	152.5581	1.3059	152.3633	1.4688
ESA	1000	292.9397	2.6246	292.0293	3.4922
SAM	1000	323.7617	2.1525	323.9805	3.3789
ESA	5000	1660.4448	3.4893	1660.0254	2.7266
SAM	5000	1795.8277	3.5084	1796.2344	5.7578
ESA	10000	3313.2738	5.5712	3313.4707	7.4297
SAM	10000	3546.1622	5.2843	3545.7656	6.7695

Query Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Peak Memory Mean (KiB)	Query Peak Memory SD (KiB)	Query Peak Memory Median (KiB)	Query Peak Memory IQR (KiB)
ESA	100	4.5784	0.0285	4.5723	0.0000
SAM	100	15.0475	0.3781	15.0781	0.5469
ESA	500	41.6934	0.0000	41.6934	0.0000
SAM	500	152.8589	1.3059	152.6641	1.4688
ESA	1000	88.8960	0.0310	88.8887	0.0000
SAM	1000	324.0625	2.1525	324.2812	3.3789
ESA	5000	476.5762	0.0000	476.5762	0.0000
SAM	5000	1796.1284	3.5084	1796.5352	5.7578
ESA	10000	962.7518	0.0697	962.7480	0.0000
SAM	10000	3546.4630	5.2843	3546.0664	6.7695

Query Extra Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Extra Memory Mean (KiB)	Query Extra Memory SD (KiB)	Query Extra Memory Median (KiB)	Query Extra Memory IQR (KiB)
ESA	100	0.5938	0.0000	0.5938	0.0000
SAM	100	0.5938	0.0000	0.5938	0.0000
ESA	500	0.6250	0.0000	0.6250	0.0000
SAM	500	0.6211	0.0000	0.6211	0.0000
ESA	1000	0.6250	0.0000	0.6250	0.0000
SAM	1000	0.6211	0.0000	0.6211	0.0000
ESA	5000	0.6250	0.0000	0.6250	0.0000
SAM	5000	0.6211	0.0000	0.6211	0.0000
ESA	10000	0.6250	0.0000	0.6250	0.0000
SAM	10000	0.6211	0.0000	0.6211	0.0000

Index Size Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Index Size Mean (KiB)	Index Size SD (KiB)	Index Size Median (KiB)	Index Size IQR (KiB)
ESA	100	9.2715	0.0000	9.2715	0.0000
SAM	100	32.8057	0.4383	32.8613	0.6562
ESA	500	44.9863	0.0000	44.9863	0.0000
SAM	500	172.9945	1.3058	172.8027	1.4688
ESA	1000	88.2754	0.0000	88.2754	0.0000
SAM	1000	342.2449	2.1524	342.4668	3.3789
ESA	5000	444.7129	0.0000	444.7129	0.0000
SAM	5000	1798.6859	3.5084	1799.0957	5.7578
ESA	10000	891.8223	0.0000	891.8223	0.0000
SAM	10000	3529.4888	5.2840	3529.0957	6.7695

Mutated Implant

Build Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Build Time Mean (ms)	Build Time SD (ms)	Build Time Median (ms)	Build Time IQR (ms)
ESA	100	0.6661	0.1265	0.6247	0.0627
SAM	100	0.2973	0.0793	0.2797	0.0240
ESA	500	27.1928	1.2921	27.2298	1.1035
SAM	500	2.3592	0.2918	2.2537	0.2469
ESA	1000	71.3355	6.8086	70.1016	3.1516
SAM	1000	5.1489	0.3975	5.0725	0.3247
ESA	5000	415.2829	23.5458	412.0689	14.8760
SAM	5000	27.3212	2.6246	27.1847	1.8712
ESA	10000	865.6591	31.8575	861.7277	26.2638
SAM	10000	58.4598	5.7681	58.5074	3.6106

Query Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Query Time Mean (ms)	Query Time SD (ms)	Query Time Median (ms)	Query Time IQR (ms)
ESA	100	0.0560	0.0146	0.0524	0.0089
SAM	100	0.0913	0.0282	0.0855	0.0164
ESA	500	1.2555	0.1061	1.2248	0.0565
SAM	500	0.4909	0.0938	0.4693	0.1025
ESA	1000	2.8705	0.2707	2.8104	0.1426
SAM	1000	1.1618	0.1691	1.1134	0.1440
ESA	5000	14.7844	0.8281	14.6585	0.6375
SAM	5000	6.0820	0.8923	5.8926	0.9030
ESA	10000	30.1891	1.1744	30.1899	1.1141
SAM	10000	13.0033	1.1745	12.9978	1.0177

Total Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Total Time Mean (ms)	Total Time SD (ms)	Total Time Median (ms)	Total Time IQR (ms)
ESA	100	0.7221	0.1348	0.6763	0.0653
SAM	100	0.3886	0.0987	0.3665	0.0330
ESA	500	28.4484	1.3192	28.4662	1.1225
SAM	500	2.8501	0.3401	2.7365	0.3251
ESA	1000	74.2060	7.0157	72.9203	3.2408
SAM	1000	6.3107	0.4960	6.2280	0.4621
ESA	5000	430.0673	24.2207	426.7509	15.3301
SAM	5000	33.4031	3.2867	33.1946	2.3144
ESA	10000	895.8482	32.6771	891.7958	27.0631
SAM	10000	71.4631	6.7327	71.5134	4.3611

Build Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Build Peak Memory Mean (KiB)	Build Peak Memory SD (KiB)	Build Peak Memory Median (KiB)	Build Peak Memory IQR (KiB)
ESA	100	17.0411	0.3534	17.0176	0.3594
SAM	100	18.6140	0.6739	18.7109	0.7344
ESA	500	117.1639	0.8303	116.9473	0.5703
SAM	500	173.6980	1.0974	173.9102	1.4492
ESA	1000	257.0182	1.8227	256.4395	3.7500
SAM	1000	379.1989	1.7619	379.3945	1.8633
ESA	5000	1398.2211	6.0046	1400.7363	2.4062
SAM	5000	2000.5287	4.5416	2000.8320	5.7969
ESA	10000	2807.9716	5.8384	2807.3301	3.4141
SAM	10000	4043.5930	5.1033	4043.3828	7.8672

Query Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Peak Memory Mean (KiB)	Query Peak Memory SD (KiB)	Query Peak Memory Median (KiB)	Query Peak Memory IQR (KiB)
ESA	100	4.6695	0.0523	4.6562	0.0605
SAM	100	19.3906	0.7071	19.4209	0.8154
ESA	500	41.7860	0.0382	41.7783	0.0488
SAM	500	174.6112	1.1849	174.5752	1.2686
ESA	1000	88.9845	0.0241	88.9756	0.0283
SAM	1000	380.2407	1.8273	380.5127	2.1729
ESA	5000	476.7691	0.1685	476.7236	0.0674
SAM	5000	2001.6002	4.5429	2001.8057	5.8643
ESA	10000	963.0100	0.1976	962.9443	0.0850
SAM	10000	4045.0428	5.3404	4045.3662	7.7617

Query Extra Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Extra Memory Mean (KiB)	Query Extra Memory SD (KiB)	Query Extra Memory Median (KiB)	Query Extra Memory IQR (KiB)
ESA	100	0.6797	0.0330	0.6777	0.0303
SAM	100	1.0971	0.3216	0.8506	0.5898
ESA	500	0.7175	0.0382	0.7100	0.0488
SAM	500	1.2336	0.3407	1.4072	0.6475
ESA	1000	0.7208	0.0241	0.7119	0.0283
SAM	1000	1.3621	0.3226	1.4932	0.6387
ESA	5000	0.8057	0.1702	0.7617	0.0527
SAM	5000	1.3918	0.2856	1.4893	0.1494
ESA	10000	0.8486	0.1877	0.7793	0.0674
SAM	10000	1.7701	0.6790	1.6328	0.2930

Index Size Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Index Size Mean (KiB)	Index Size SD (KiB)	Index Size Median (KiB)	Index Size IQR (KiB)
ESA	100	9.2715	0.0000	9.2715	0.0000
SAM	100	36.9114	0.7756	37.0273	0.8281
ESA	500	44.9863	0.0000	44.9863	0.0000
SAM	500	193.7652	1.0974	193.9805	1.4492
ESA	1000	88.2754	0.0000	88.2754	0.0000
SAM	1000	397.3130	1.7619	397.5117	1.8633
ESA	5000	444.7129	0.0000	444.7129	0.0000
SAM	5000	2003.0178	4.5416	2003.3242	5.7969
ESA	10000	891.8223	0.0000	891.8223	0.0000
SAM	10000	4026.5508	5.1033	4026.3438	7.8672

Near Identical

Build Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Build Time Mean (ms)	Build Time SD (ms)	Build Time Median (ms)	Build Time IQR (ms)
ESA	100	0.8973	0.2670	0.8299	0.0721
SAM	100	0.2219	0.0571	0.2083	0.0194
ESA	500	28.7797	3.8677	28.6069	1.2660
SAM	500	1.7730	0.3135	1.6721	0.1911
ESA	1000	72.0393	3.9829	71.6989	3.5016
SAM	1000	4.0787	0.3182	4.0387	0.2440
ESA	5000	436.3448	32.8775	429.1522	15.7043
SAM	5000	20.1832	2.5592	19.9330	1.6187
ESA	10000	917.2661	32.8563	914.5540	27.2977
SAM	10000	43.1988	2.6354	43.3967	2.8946

Query Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Query Time Mean (ms)	Query Time SD (ms)	Query Time Median (ms)	Query Time IQR (ms)
ESA	100	0.0592	0.0214	0.0548	0.0095
SAM	100	0.3424	0.0579	0.3204	0.0285
ESA	500	1.3001	0.1472	1.2532	0.0632
SAM	500	2.5607	0.1944	2.5114	0.1640
ESA	1000	2.9253	0.3626	2.8856	0.1385
SAM	1000	6.1874	0.4308	6.1105	0.4002
ESA	5000	15.5071	1.1774	15.2848	0.6238
SAM	5000	35.6452	4.7441	35.0750	1.5872
ESA	10000	31.5783	1.9220	31.4201	1.1969
SAM	10000	77.5589	3.7466	77.3678	2.9818

Total Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Total Time Mean (ms)	Total Time SD (ms)	Total Time Median (ms)	Total Time IQR (ms)
ESA	100	0.9566	0.2782	0.8846	0.0788
SAM	100	0.5643	0.1049	0.5343	0.0458
ESA	500	30.0799	3.9293	29.9108	1.2967
SAM	500	4.3337	0.4158	4.2065	0.4002
ESA	1000	74.9646	4.0878	74.6613	3.7002
SAM	1000	10.2660	0.6157	10.1860	0.5347
ESA	5000	451.8520	33.8624	444.4871	16.2480
SAM	5000	55.8285	6.4827	54.9929	2.8518
ESA	10000	948.8444	34.0679	946.0367	28.3186
SAM	10000	120.7577	6.1184	120.8392	5.2513

Build Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Build Peak Memory Mean (KiB)	Build Peak Memory SD (KiB)	Build Peak Memory Median (KiB)	Build Peak Memory IQR (KiB)
ESA	100	19.7698	0.4242	19.7598	0.6875
SAM	100	13.7309	0.5255	13.9219	0.9219
ESA	500	122.9600	0.8263	123.2051	0.8828
SAM	500	142.9375	1.2450	142.9375	1.8828
ESA	1000	268.5878	1.5279	268.3926	1.0078
SAM	1000	322.5197	1.4953	322.2227	1.6875
ESA	5000	1507.8776	11.8780	1509.4785	18.1016
SAM	5000	1645.8894	3.1171	1644.9648	4.7539
ESA	10000	3048.2410	11.0266	3046.9082	10.7422
SAM	10000	3383.0981	4.6633	3383.0508	7.2930

Query Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Peak Memory Mean (KiB)	Query Peak Memory SD (KiB)	Query Peak Memory Median (KiB)	Query Peak Memory IQR (KiB)
ESA	100	4.8014	0.0496	4.7881	0.0020
SAM	100	14.3064	0.5256	14.4971	0.9219
ESA	500	50.0279	0.0776	49.9873	0.0654
SAM	500	144.3398	1.2458	144.3369	1.8926
ESA	1000	113.4915	0.0844	113.4805	0.0664
SAM	1000	324.8686	1.4894	324.5850	1.7168
ESA	5000	631.4863	0.3430	631.3613	0.2324
SAM	5000	1655.9121	3.1172	1654.9795	4.7598
ESA	10000	1280.3577	0.6053	1280.2090	0.0791
SAM	10000	3402.6903	4.6552	3402.6357	7.2949

Query Extra Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Extra Memory Mean (KiB)	Query Extra Memory SD (KiB)	Query Extra Memory Median (KiB)	Query Extra Memory IQR (KiB)
ESA	100	0.8066	0.0093	0.8096	0.0010
SAM	100	0.8958	0.0007	0.8955	0.0000
ESA	500	1.6002	0.0159	1.6064	0.0078
SAM	500	1.7226	0.0043	1.7197	0.0039
ESA	1000	2.5558	0.0223	2.5635	0.0039
SAM	1000	2.6792	0.0028	2.6787	0.0039
ESA	5000	10.2275	0.0206	10.2217	0.0068
SAM	5000	10.3430	0.0195	10.3350	0.0059
ESA	10000	19.7919	0.0430	19.7900	0.0029
SAM	10000	19.9125	0.0334	19.9033	0.0020

Index Size Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Index Size Mean (KiB)	Index Size SD (KiB)	Index Size Median (KiB)	Index Size IQR (KiB)
ESA	100	9.2715	0.0000	9.2715	0.0000
SAM	100	32.1459	0.5796	32.3750	0.9727
ESA	500	51.4252	0.0605	51.3848	0.0547
SAM	500	163.9070	1.2450	163.9102	1.8828
ESA	1000	108.1062	0.0389	108.0996	0.0547
SAM	1000	341.5261	1.4898	341.2422	1.7227
ESA	5000	571.8373	0.2732	571.7246	0.0820
SAM	5000	1649.2808	3.1171	1648.3594	4.7539
ESA	10000	1152.9190	0.4672	1152.7910	0.0273
SAM	10000	3366.9583	4.6633	3366.9141	7.2930

Random Uniform

Build Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Build Time Mean (ms)	Build Time SD (ms)	Build Time Median (ms)	Build Time IQR (ms)
ESA	100	0.4797	0.0980	0.4476	0.0406
SAM	100	0.2277	0.0712	0.2094	0.0250
ESA	500	25.7370	4.2493	25.5164	1.1177
SAM	500	1.7637	0.2669	1.6816	0.1561
ESA	1000	67.5274	3.3034	66.9891	3.0356
SAM	1000	4.1209	0.4263	4.0545	0.2428
ESA	5000	396.0339	14.3132	393.8816	16.4700
SAM	5000	20.1749	2.9370	20.1750	1.4106
ESA	10000	832.9461	32.7767	828.3093	23.6200
SAM	10000	43.4903	4.1890	43.5015	2.2644

Query Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Query Time Mean (ms)	Query Time SD (ms)	Query Time Median (ms)	Query Time IQR (ms)
ESA	100	0.0618	0.0136	0.0604	0.0083
SAM	100	0.0712	0.0235	0.0675	0.0143
ESA	500	1.3919	1.4714	1.2696	0.1225
SAM	500	0.4728	0.0816	0.4555	0.0810
ESA	1000	2.9215	0.2562	2.8963	0.2120
SAM	1000	1.0571	0.1328	1.0295	0.1642
ESA	5000	14.9142	0.5880	14.8299	0.6740
SAM	5000	5.9725	1.1348	5.8729	0.7757
ESA	10000	30.4111	1.4659	30.2856	0.9637
SAM	10000	12.1748	0.9740	12.1457	0.7932

Total Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Total Time Mean (ms)	Total Time SD (ms)	Total Time Median (ms)	Total Time IQR (ms)
ESA	100	0.5415	0.1048	0.5071	0.0467
SAM	100	0.2988	0.0874	0.2758	0.0301
ESA	500	27.1289	4.5455	26.8161	1.1529
SAM	500	2.2365	0.3149	2.1447	0.2199
ESA	1000	70.4489	3.4065	69.8890	3.1449
SAM	1000	5.1780	0.4905	5.1077	0.3604
ESA	5000	410.9481	14.6772	408.7129	16.9720
SAM	5000	26.1474	3.1944	25.9976	1.4997
ESA	10000	863.3572	33.8125	858.3702	24.3951
SAM	10000	55.6651	4.9964	55.7341	2.7074

Build Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Build Peak Memory Mean (KiB)	Build Peak Memory SD (KiB)	Build Peak Memory Median (KiB)	Build Peak Memory IQR (KiB)
ESA	100	16.0465	0.5712	16.0645	0.4141
SAM	100	13.7722	0.9011	14.0078	1.1016
ESA	500	125.2396	0.5148	125.1348	0.8750
SAM	500	142.9797	1.2711	142.7109	0.8086
ESA	1000	298.8832	3.6421	297.7715	5.8906
SAM	1000	323.2071	1.7084	322.6719	2.2891
ESA	5000	1740.3898	10.5886	1742.2676	18.3594
SAM	5000	1645.7923	3.2911	1646.0859	3.6875
ESA	10000	3500.5570	6.7340	3501.4941	9.6562
SAM	10000	3385.0009	4.8256	3385.0625	5.7070

Query Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Peak Memory Mean (KiB)	Query Peak Memory SD (KiB)	Query Peak Memory Median (KiB)	Query Peak Memory IQR (KiB)
ESA	100	5.3109	0.2870	5.3125	0.2100
SAM	100	14.9186	0.9756	15.1055	1.2656
ESA	500	43.1870	1.1188	42.7129	0.2988
SAM	500	145.6341	3.1733	144.6836	2.4854
ESA	1000	91.2228	1.4616	91.9395	2.9180
SAM	1000	325.9029	1.8814	325.8105	2.1445
ESA	5000	479.0084	1.3701	479.6514	2.4336
SAM	5000	1649.3546	4.2902	1649.0820	3.6553
ESA	10000	965.8775	2.3907	965.8662	2.6777
SAM	10000	3389.3315	5.8528	3388.2588	7.0273

Query Extra Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Extra Memory Mean (KiB)	Query Extra Memory SD (KiB)	Query Extra Memory Median (KiB)	Query Extra Memory IQR (KiB)
ESA	100	1.3143	0.2728	1.3340	0.2100
SAM	100	1.4667	0.3255	1.4199	0.2119
ESA	500	2.1186	1.1188	1.6445	0.2988
SAM	500	2.9747	2.4931	1.7979	2.0547
ESA	1000	2.9540	1.4624	3.6758	2.9180
SAM	1000	3.0162	1.3717	3.7891	2.4268
ESA	5000	3.0573	1.3701	3.7002	2.4336
SAM	5000	3.8826	3.1630	3.7861	2.4883
ESA	10000	3.7357	2.3906	3.7432	2.6309
SAM	10000	4.6509	4.0120	3.8291	2.8184

Index Size Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Index Size Mean (KiB)	Index Size SD (KiB)	Index Size Median (KiB)	Index Size IQR (KiB)
ESA	100	9.2715	0.0000	9.2715	0.0000
SAM	100	32.1570	0.7439	32.4609	1.2520
ESA	500	44.9863	0.0000	44.9863	0.0000
SAM	500	163.9492	1.2710	163.6836	0.8086
ESA	1000	88.2754	0.0000	88.2754	0.0000
SAM	1000	342.2234	1.7084	341.6914	2.2891
ESA	5000	444.7129	0.0000	444.7129	0.0000
SAM	5000	1649.1837	3.2911	1649.4805	3.6875
ESA	10000	891.8223	0.0000	891.8223	0.0000
SAM	10000	3368.8611	4.8256	3368.9258	5.7070

Repetitive with Noise

Build Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Build Time Mean (ms)	Build Time SD (ms)	Build Time Median (ms)	Build Time IQR (ms)
ESA	100	0.7456	0.3978	0.7453	0.2040
SAM	100	0.3124	0.0942	0.3029	0.0544
ESA	500	28.8745	4.1395	28.5099	1.6150
SAM	500	2.6619	0.7030	2.5061	0.3283
ESA	1000	75.6570	10.9879	72.4695	6.3244
SAM	1000	5.7827	1.5698	5.5807	0.5132
ESA	5000	420.6713	18.8932	417.7870	17.4356
SAM	5000	29.8320	2.4029	29.6727	1.8447
ESA	10000	880.0391	33.7104	873.1154	35.8926
SAM	10000	63.0692	3.5125	63.2876	4.3649

Query Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Query Time Mean (ms)	Query Time SD (ms)	Query Time Median (ms)	Query Time IQR (ms)
ESA	100	0.0667	0.0184	0.0637	0.0213
SAM	100	0.1309	0.0288	0.1259	0.0272
ESA	500	1.2991	0.2699	1.2407	0.0803
SAM	500	0.5194	0.0943	0.4999	0.0738
ESA	1000	2.9297	0.3893	2.8582	0.2007
SAM	1000	1.2252	1.0578	1.1130	0.1993
ESA	5000	14.7592	0.6385	14.6447	0.5938
SAM	5000	5.6016	1.0986	5.4534	0.7179
ESA	10000	30.1369	1.2091	30.0663	1.0998
SAM	10000	11.9808	0.7847	11.9413	0.9435

Total Time Comparison

Based on 500 benchmark runs per input length. Times in milliseconds; lower is better.

Algorithm	Length	Total Time Mean (ms)	Total Time SD (ms)	Total Time Median (ms)	Total Time IQR (ms)
ESA	100	0.8122	0.4013	0.8030	0.2064
SAM	100	0.4433	0.1109	0.4202	0.0666
ESA	500	30.1736	4.2170	29.8178	1.6146
SAM	500	3.1813	0.7532	3.0233	0.4095
ESA	1000	78.5866	11.1725	75.3565	6.5258
SAM	1000	7.0080	2.0697	6.7805	0.6139
ESA	5000	435.4306	19.3675	432.2704	17.8116
SAM	5000	35.4336	3.2577	35.2052	2.1612
ESA	10000	910.1760	34.5820	903.2966	36.6330
SAM	10000	75.0500	4.1045	75.4318	4.7936

Build Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Build Peak Memory Mean (KiB)	Build Peak Memory SD (KiB)	Build Peak Memory Median (KiB)	Build Peak Memory IQR (KiB)
ESA	100	16.9204	2.1218	15.2129	4.0312
SAM	100	20.3926	3.3077	21.0469	3.7109
ESA	500	116.5432	3.3041	116.4941	6.2852
SAM	500	191.3998	6.0547	191.8555	6.8320
ESA	1000	258.3808	4.4882	259.4238	5.1797
SAM	1000	412.5785	8.4709	411.0703	14.2461
ESA	5000	1388.0634	13.1931	1392.7285	22.5469
SAM	5000	2197.5287	9.6697	2196.5898	14.0781
ESA	10000	2806.4719	21.2297	2796.6348	38.2422
SAM	10000	4435.5413	14.3320	4439.6953	18.6328

Query Peak Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Peak Memory Mean (KiB)	Query Peak Memory SD (KiB)	Query Peak Memory Median (KiB)	Query Peak Memory IQR (KiB)
ESA	100	4.6909	0.0542	4.6641	0.0576
SAM	100	20.8789	3.3049	21.4814	3.6904
ESA	500	41.8202	0.0377	41.8115	0.0215
SAM	500	191.9590	6.0486	192.3955	6.8730
ESA	1000	89.0627	0.1435	89.0342	0.0586
SAM	1000	413.2130	8.4098	411.5576	14.3164
ESA	5000	476.7843	0.0511	476.7861	0.0742
SAM	5000	2198.2025	9.6755	2197.3115	14.0137
ESA	10000	962.9840	0.0635	962.9619	0.0811
SAM	10000	4436.2402	14.3303	4440.3730	18.4805

Query Extra Memory Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Query Extra Memory Mean (KiB)	Query Extra Memory SD (KiB)	Query Extra Memory Median (KiB)	Query Extra Memory IQR (KiB)
ESA	100	0.6961	0.0276	0.6846	0.0459
SAM	100	0.8068	0.0302	0.8096	0.0410
ESA	500	0.7517	0.0374	0.7432	0.0215
SAM	500	0.8795	0.0514	0.8643	0.0586
ESA	1000	0.7905	0.0533	0.7705	0.0586
SAM	1000	0.9550	0.1439	0.9248	0.0527
ESA	5000	0.8204	0.0452	0.8330	0.0781
SAM	5000	0.9941	0.0941	0.9775	0.0352
ESA	10000	0.8255	0.0551	0.8008	0.0801
SAM	10000	1.0192	0.1121	1.0068	0.0352

Index Size Comparison

Based on 500 benchmark runs per input length. Memory in KiB; lower is better.

Algorithm	Length	Index Size Mean (KiB)	Index Size SD (KiB)	Index Size Median (KiB)	Index Size IQR (KiB)
ESA	100	9.2715	0.0000	9.2715	0.0000
SAM	100	38.9295	3.7425	39.7188	4.2168
ESA	500	44.9863	0.0000	44.9863	0.0000
SAM	500	211.4670	6.0547	211.9258	6.8320
ESA	1000	88.2754	0.0000	88.2754	0.0000
SAM	1000	430.6920	8.4712	429.1875	14.2461
ESA	5000	444.7129	0.0000	444.7129	0.0000
SAM	5000	2200.0178	9.6697	2199.0820	14.0781
ESA	10000	891.8223	0.0000	891.8223	0.0000
SAM	10000	4418.4991	14.3320	4422.6562	18.6328

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
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

System	Prompt	Usage
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
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

System	Prompt	Usage
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
GitHub Copilot 25	Reported that generated benchmark tables do not fit on the page and requested a smart code-side solution (splitting tables if needed) instead of shrinking them.	Updated the table-export generator to split each algorithm summary into multiple narrower tables (timing, memory, index size)
GitHub Copilot 26	Requested that the generated benchmark tables should not show up in the List of Tables.	Adjusted the generator to avoid captions and instead use a custom numbered title with `\refstepcounter{table}` and `\label` for cross-references
GitHub Copilot 27	Requested that the generated tables are still slightly too large and should use a smaller font size.	Wrapped generated tabular blocks in `\small` within the generator to reduce width without making tables unreadably tiny

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 28, 2026 ....

Rubin Chempananickal James