Optimizing Text Validation Using Trie Data Structures in Typing Games: An Algorithmic Complexity Analysis

Mirza Aryasatya Akmal - 13525131
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: mirzaaryast14@gmail.com, 13525131@std.stei.itb.ac.id

***Abstract*—Nowadays, interactive text softwares including text editors, autocomplete features, and input/command driven applications like typing games such as *Monkeytype* and *Typeracer* require real time text validation. In such systems, user inputs must be verified against a quite sizable lexicon instantly in order to maintain a satisfactory response rate from the system. This paper aims to comparatively analyze the algorithmic complexity and computational efficiency of traditional linear array searches against the Trie data structure. Through multi scale benchmarking using Python, the Trie structure returned significant performance optimization, reducing the search complexity from $O(N \times M)$ to $O(M)$ and stabilizing the lookup latency down to under 1 ms across an 80.000 word dataset. However, this acceleration in speed obeys a certain space-time tradeoff with a larger memory footprint of roughly 68 MB ($O(\Sigma \times N \times M)$) compared to the array's 0.69 MB ($O(N \times M)$). Results yielded from benchmarking proved that using a tree based approach effectively can mitigate processing delays, making sure as little as milliseconds response time regardless of the lexicon sizes stays consistent.**

Keywords—Trie, Algorithmic Complexity, Tree Theory, Text Validation, Linear Search

I. Introduction

In modern software engineering, real time string processing and text validation often carry real limitations in the face of complexity. Interactive softwares and web applications alike can demand high levels of responsiveness and immediate feedback mechanisms to the user in which every key press and/or completed words need to be instantaneously cross referenced with certain predefined dictionaries or lexicons already stored in memory. An example of this can be found in interactive typing web applications/games and/or command line interfaces, such as in popular competitive typing platforms *Monkeytype* and *Typeracer*. These applications demand very low latency, where *Monkeytype* requires character-by-character validation in order to instantaneously highlight user's errors at any time while calculating their Words Per Minute (WPM), and *Typeracer* mandates a synchronized text validation method across multiple players in a single multiplayer session. In both cases, the underlying software will have to validate the user's string inputs against possibly thousands of words within milliseconds to prevent any impairing input lag and maintaining a streamlined user experience. This validation mechanism is different from simple, stateless string matching of independent words. Rather than waiting for a word boundary to trigger verification, typing applications/games require continuous prefix evaluation at keystroke levels, ensuring that any target miss from the lexicon is detected instantly.

The standard and most intuitive way to manage lexicons/dictionaries is to store data within sequential arrays or lists. In order to validate a word, the system will perform a linear search, scanning sequentially until a match is either found or the end of the array is reached. While it is straightforward and simple to implement, this method has severe limitations in terms of scale where if the lexicon were to grow, the complexity increases proportionally and therefore leads to noticeable delays during high frequency inputs. For input applications such as from professional typists regularly exceeding 150 WPM, the traditional sequential scanning method can trigger computational delays where the system interface lags far behind the user's physical inputs.

To resolve this complexity issue, discrete mathematics provides a more efficient alternative through the form of Tree Theory, specifically through the implementation of the retrieval tree or the Trie data structure. Theoretically, a Trie restructures string storage by mapping commonly used prefixes to certain shared nodes, making the search time entirely independent of the dictionary/lexicon's total word count and is strictly dependant on the length of the queried string. This paper demonstrates the practical application of Trie structures and comparing its computational behaviour with the traditional linear data structures through testing.

II. Theoretical Foundation

A. Graph and Tree Theory

In discrete mathematics, a graph is a structure used to model pairwise relations between objects. A graph $G = (V, E)$ consists of vertices V and edges E that connect

these vertices. A tree is a type of connected graph which does not contain a circuit, and has 1 node that functions as a root. The structure of a rooted tree includes several terminologies:

1. Node/Vertex: The structural elements which store data or represent a certain state.
2. Edge: The link connecting a parent node/vertex to a child node/vertex.
3. Parent and Child: If a directed edge runs from node u to v then u is the parent of v, and v is the child of u.
4. Leaf: A node with no children.
5. Path: A sequence of edges that connects a sequence of nodes, starting from the root into a certain descendant node [1].

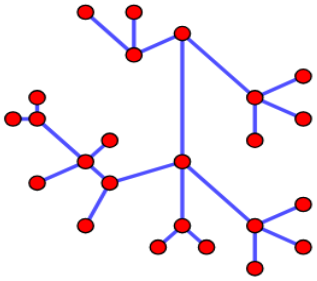


Fig. 1. Example of a Tree

(Taken from https://www.mathreference.org/index/page/id/393/lg/en)

B. *Trie Data Structure*

A Trie, also referred to as a prefix tree, is an ordered tree-based data structure that is used to store an associative array. Unlike a standard binary search tree, no node in the trie stores the key associated with that node. Instead, its position in the trie defines the key which it is associated with. All of the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string [2][3].

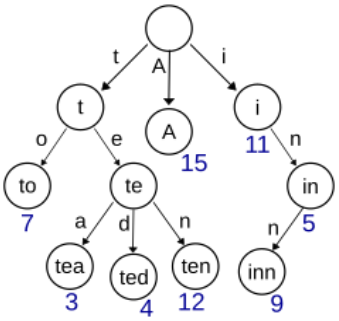


Fig. 2. Example of a Trie
Taken from (https://en.wikipedia.org/wiki/Trie)

The structure of a single Trie node comprises two main components:

1. A Map/Dictionary of Children: A collection of links to child nodes where each key corresponds into a certain character of the alphabet.
2. An End-of-Word Flag: A boolean indicator (such as *is_end_of_word*) to determine whether a path from the root to a certain node fulfills a complete and/or valid word within the lexicon/dictionary, rather than just a prefix.

When inserting a string of length M into a trie, the algorithm begins at the root of the trie and traverses downward, following the characters of the string. If a character path does not exist, a new child node is instantiated. Upon reaching the final character, the end-of-word flag is then set to true. The search process then follows an identical traversal path so that if a character path is missing or if the final node's flag is false when the string character inputs have ended, the string does not exist in the lexicon/dictionary.

C. *Algorithmic Complexity and Big-O Notation*

Algorithmic complexity is used to quantify the amount of time and/or memory space required by a certain algorithm to run as a function of the input size. Time complexity in algorithmic complexity is commonly expressed using Big-O notation, which shows the limit of a function when a certain argument goes to infinity [4].

In the relevant context of text validation against a predefined lexicon/dictionary, there are two lookup/searching mechanisms to be evaluated:

1. Linear Array Search: Storing a lexicon/dictionary containing N amount of unique words in a sequential array requires also a sequential scan during the validation process. In order to verify if an input string of length M exists, the system then compares the input against each element in the array. In the worst-case scenario, where the word at the end of the array does not exist, the algorithm performs N amount of comparisons, with each string comparison taking up to $O(M)$ time. Therefore, the total time complexity for a linear array search is noted by:

$$T(N, M) = O(N \times M)$$

2. Trie Lookup: In a Trie data structure, the search operation only requires the algorithm to traverse down the tree branches character by character based on the input string. The algorithm performs exactly 1 operation per character. Therefore, the search time is entirely independent of the total number of words stored in the dictionary. The worst-case time complexity for the Trie lookup is determined strictly by the length of the query string following:

$$T(M) = O(M)$$

D. Real-Time Text Validation

Real-time text validation is the process of verifying string inputs simultaneously with the user's interactions at any given time. In order to maintain a responsive interface with a fast and smooth input latency from the user, any searching algorithm must execute within milliseconds of thresholds or less. If the validation time scales with the database size, fast inputs in typing games from the user such as WPM counts exceeding 150 will trigger a queue backlog, therefore causing noticeable and physically hindering input lag [5].

Text validation in large string dictionaries shows a matching problem especially compared to simple text search operations. Traditional string matching algorithms use a stateless string in their inputs, therefore repeating comparisons over and over again from the first character after every modification up until the last keystroke. By using Trie structure, text validation will become stateful. Systems can now have constant reference to the currently active node in the graph, making keystroke validations from regular string comparisons to localized node transitions.

E. Space Complexity

Analyzing space complexity is important in understanding the trade off between computing space and time which inevitably will happen in the search based trie data structure. While the Trie data structure theoretically offers a much higher time efficiency in the case of text/string validation because of its independence from the lexicon/dictionary size, it comes at the price of a higher memory consumption compared to a simple linear array search.

In a linear array, a collection of string/words is stored sequentially in memory. If there exists N amount of words with the average length of M characters, then the space complexity needed is linear towards the total amount of characters which is noted by:

$$S(N, M) = O(N \times M)$$

On the contrary, every node in a standard Trie structure (R-way Trie) must store a pointer array/dictionary into every possible next child node. The size of this pointer array is designated by the size of the alphabet Σ that is used (e.g Σ = 26 to represent the lowercase a-z in the latin alphabet). Therefore if a structure has a total V amount of vertices, the total space complexity of the Trie structure is $O(\Sigma \times V)$. In a worst-case scenario where no inputted words share an alphabet sequence or the same prefix, the amount of vertices will reach its peak, with a space complexity of:

$$S(\Sigma, N, M) = O(\Sigma \times N \times M)$$

The Σ as a multiplier factor is what causes the Trie structure to consume a lot more memory (space) compared to conventional linear arrays [3]. However, in the case of modern interactive systems, the usage of way more memory is more than worth it and is widely implemented to get a desirably less user response time.

III. Implementation

A. Experimental Setup and Environment

To compare the time and space complexity of the conventional linear array search against the Trie structure search, a Python program is used to simulate conditions similar to the aforementioned *Monkeytype* application and *Typeracer* web game.

To ensure thoroughness, precision and to reduce as many external factors during the performance evaluation, a benchmarking simulation is executed under a certain computing environment. The hardware configuration used to test the algorithmic complexity is an Intel Core i7-14700K processor at stock frequencies paired with 32 GB of DDR5 RAM. The testing platform was devised using Python (Version 3.13.7) without external web dependencies or any game engine frameworks to purely isolate the core string/text validation algorithms from other graphic rendering or framework related overheads found in the real web application/game.

The experiment design uses generated datasets to represent text lexicons of escalating scales. The dictionaries used consist of distinct string tokens generated from the standard lowercase latin alphabet (a-z, Σ = 26), with individual word lengths specifically constrained between 5 and 8 characters to simulate standard lexical distributions in real text softwares.

B. Algorithmic Implementation

The aforementioned computational benchmark directly compares the two distinct paradigms of string/text organization and search/lookup mechanics.

1. Linear Array Search: The control group uses Python's built in dynamic array/list structure. The validation function will perform an unoptimized sequential scan across the string/text collection. The iteration will terminate after finding an exact structural match or if the boundary of the dataset is reached.

```python
# Linear Search Algorithm For Text Validation
def linear_search(array_list, word):
    for item in array_list:
        if item == word:
            return True
    return False
```

Fig. 3. Linear Array Search For Text Validation

2. Trie Structure: The experimental group relies on an object oriented implementation of a prefix tree. The

structural hierarchy is defined by 2 classes: TrieNode and Trie. The node object maps each individual character transformations using a native dictionary hash map while maintaining a boolean status to indicate a path completion.

```python
# Trie Data Structure For Text Validation
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
```

Fig. 4. Trie Structure For Text Validation

C. Multi-Scale Benchmarking Results

To gain the results needed to compare both algorithms in terms of their algorithmic complexity, the following functions are used.

```python
def generate_random_word(min_len=5, max_len=8):
    length = random.randint(min_len, max_len)
    return ''.join(random.choices(string.ascii_lowercase, k=length))

def generate_dataset(size):
    lexicon = set()
    while len(lexicon) < size:
        lexicon.add(generate_random_word())
    return list(lexicon)
```

Fig. 5. Word and Dataset Generation Functions

Functions generate_random_word and generate_dataset are used to prepare both algorithms in the next run_benchmarking function.

```python
def run_benchmark():
    dataset_sizes = [5000, 10000, 20000, 40000, 80000]
    total_queries = 1000
    print(f"{'Lexicon Size (N)':<18} | {'Linear Search (ms)':<20} | {'Trie Lookup (ms)':<18} | {'Speedup'}")
    for size in dataset_sizes:
        lexicon_list = generate_dataset(size)
        trie = Trie()
        for word in lexicon_list:
            trie.insert(word)
        queries = []
        queries.extend(random.choices(lexicon_list, k=total_queries // 2))
        for _ in range(total_queries // 2):
            queries.append(generate_random_word(9, 12))
        random.shuffle(queries)

        start_linear = time.perf_counter()
        for q in queries:
            linear_search(lexicon_list, q)
        end_linear = time.perf_counter()
        time_linear_ms = (end_linear - start_linear) * 1000

        start_trie = time.perf_counter()
        for q in queries:
            trie.search(q)
        end_trie = time.perf_counter()
        time_trie_ms = (end_trie - start_trie) * 1000

        speedup = time_linear_ms / time_trie_ms if time_trie_ms > 0 else 0

        print(f"{size:<18} | {time_linear_ms:<20.2f} | {time_trie_ms:<18.2f} | {speedup:.2f}x")
```

Fig. 6. Run Time Complexity Benchmark Function For Both Text Validation Algorithms

In order to stress-test the dictionary scale limits based on asymptotic complexity theories, both algorithms are subjects to a multiscale benchmarking process. The validation pool size for each test is consistently maintained at 1000 query operations with 50% split between proven valid words and randomized invalid strings/texts. The latency is then measured in milliseconds using Python's time.perf_counter() module. The results across 5 independent lexical scales are as follows.

Lexicon Size (N)	Evaluation Queries	Linear Array Search Latency (ms)	Trie Tree Lookup Latency (ms)	Acceleration Factor (×)
5000	1000	39.19	0.58	82.17×
10000	1000	77.82	0.61	126.74×
20000	1000	160.21	0.71	226.32×
40000	1000	316.95	0.84	376.69×
80000	1000	635.28	0.94	672.40×

TABLE 1. MULTI-SCALE PERFORMANCE BENCHMARK RESULTS

The dataset above shows a direct alignment with theoretical complexity limits. For the linear array search, latency scales linearly relative to the dictionary volume (N). At a dictionary size of 5000 words, the sequential lookup finishes in 39.19 ms. However as the dictionary grows to 80000 words, the latency spikes up to 635,28 ms. This behaviour describes the realities of the $O(N \times M)$ worst case boundary where expanding the dictionary forces linearly expanding comparison operations for the processor.

In contrast to the linear array search, the Trie lookup performance while not remaining flat, maintains a constant latency of less than 1 ms up until the 80000 dictionary words mark. This proves that the algorithm follows the $O(M)$ complexity profile of the Trie structure.

D. Space Complexity Evaluation

While the Trie structure achieves remarkably higher performance in terms of latency over the linear array search, it also operates on a strict space-time tradeoff principle. In order to quantify the physical memory overhead, a memory allocation benchmark is conducted using Python's native tracemalloc utility. The following program is used to measure each algorithm's memory allocations with the same Linear Array Search and Trie Data Structure defined from before.

```
if __name__ == "__main__":
    dataset_size = 80000
    print(f"Generating {dataset_size} words...")
    words = [generate_random_word() for _ in range(dataset_size)]

    tracemalloc.start()
    lexicon_list = []
    for w in words:
        lexicon_list.append(w)

    _, peak_array = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    array_memory_kb = peak_array/1024
    print(f"Linear Array Search Memory : {array_memory_kb:.2f} KB")
    tracemalloc.start()
    trie = Trie()
    for w in words:
        trie.insert(w)

    _, peak_trie = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    trie_memory_kb = peak_trie / 1024
    print(f"Trie Memory: {trie_memory_kb:.2f} KB")

    ratio = trie_memory_kb/array_memory_kb
    print(f"Ratio TDS:LAS {ratio:.2f} : 1")
```

Fig. 7. Malloc Benchmark Program for Both Text Validation Algorithms

The following table is the results of the memory allocation footprints of each algorithm.

Data Structure	Lexicon Size (N)	Memory Consumption	Theoretical Space Complexity
Linear Array	80.000	0.695 MB	$O(N \times M)$
Trie	80.000	68.657 MB	$O(\Sigma \times N \times M)$

TABLE 2. MEMORY ALLOCATION FOOTPRINT (SPACE COMPLEXITY)

The footprint results clarify the overhead brought by the alphabet multiplier ($\Sigma = 26$). The linear array only needed a mere 0.695 MB of memory allocation whereas the Trie structure demands 68.657 MB of memory allocation to house a similarly sized 80.000 word dictionary. This significance in size difference is due to each independent node in a standard R-way Trie instantiating a full pointer dictionary instance. Despite using much more memory, a memory footprint of 68 MB can still be considered trivial for modern computer architectures. In the design of a fast-paced interactive typing applications/software, allocating a substantial amount of RAM to reassure an absolute $O(M)$ validation response time is an optimal engineering decision.

E. Computational Execution Trace

The structural optimization of the Trie structure is best understood by tracing the state transformations of each letter within the memory graph. Consider an empty Trie as a subject to the sequential insertion of 3 target lexical tokens: "type", "typeracer", and "monkey".

1. Insertion of "type": Starting at the root node, system detects an empty child map, instantiates 4 consecutive descending nodes connected by directed character edges: (Root)→[t]→[y]→[p]→[e]. The node corresponding to [e] will have its internal flag toggled to denote the end of a valid word.

2. Insertion of "typeracer": After evaluating characters t, y, p, and e, the algorithm recognizes the prefix path already exists in the memory. Rather than allocating new objects, the algorithm traverses the existing node [e], and then branches out to instantiate 5 new child nodes representing the suffix: [r]→[a]→[c]→[e]→[r].

3. Insertion of "monkey": Because the memory lacks a matching edge at the root node for the character m, system initializes a new path branch: (Root)→[m]→[o]→[n]→[k]→[e]→[y].

When a search query for "typeracer" is executed, the algorithm traverses down the shared prefix path without scanning any unrelated branches such as the "monkey" branch. In practice the search operation will avoid checking thousands of unrelated words, completing the validation in a minimal number of steps.

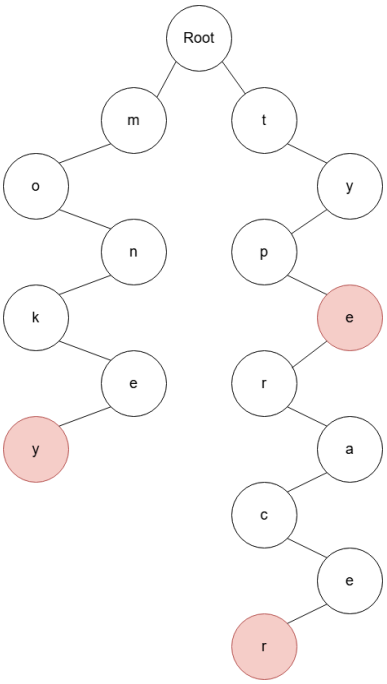


Fig. 8. Trie Structure Inserted with Lexical Tokens “type”, “typeracer”, and “monkey”

F. Comparative Application Analysis: Keystroke-Level against Word-Level Mechanics

The differences in performance from both algorithms shows certain characteristics when making interactive, text validation systems such as web typing games *Typeracer* or *Monkeytype*.

1. Word Level Validation: In applications similar to *Typeracer*, validation operations occur discretely at word boundaries, typically triggered when a user hits the spacebar key. If the validation engine is processing incoming words from multiple concurrent players over a network socket, a linear search will block the main execution thread. This delay will create a processing backlog that will degrade system updates.

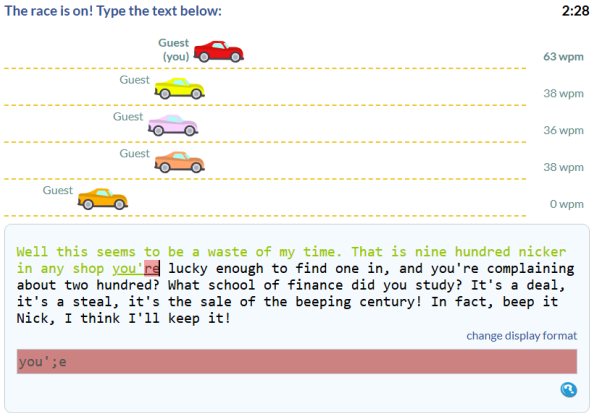


Fig. 9. Text Validation at Every Word Boundary for Multiplayer in Typeracer

2. Keystroke Level Validation: In applications reflecting the mechanism of *Monkeytype*, the advantages of the tree model becomes more critical in ensuring an instant, character-by-character text/string validation to provide instant visual feedback. If a linear array search is used for a user typing even at 120 words per minute, the system must then perform a full array scan hundreds of times per minute, therefore causing noticeable and hindering input lag.

This optimization also reveals why a Trie doesn’t just match individual words, but also traces the computational states. While a linear search algorithm forces the processor to execute a stateless $O(N \times M)$ array scans on every keystroke, the Trie uses pointer dictionaries to get a stateful $O(M)$ complexity per keystroke. The application also doesn’t reevaluate the entire string, instead it validates the immediate directed edge of the current node, therefore completely isolating processing overheads from both the dictionary size and the word length.

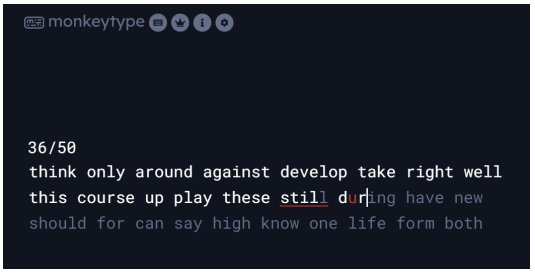


Fig. 10. Monkeytype Utilizes Keystroke Level Validation to Determine User WPM

G. Benchmarking Methodology Limitations

While the results established the benefits of the Trie structure in time complexity, the benchmarking method itself inherently has some limitations that affects the findings:

1. Synthetic Lexicon Distribution: The benchmark uses a synthetically generated lexicon composed of characters which are randomized uniformly. However, in natural languages such as English or Indonesian, word distributions tend to skew toward specific root words and many shared prefixes (e.g “un-”, “re-”, “be-”, “me-”, “ber-”, etc.). A natural language dataset would result in a more dense shared prefix nodes near the Trie root which would affect the structural shape and memory footprint compared to the randomized model used in the benchmarking.

2. High Level Language Usage: The benchmarking was done purely using the Python programming language which is a high level language. In Python, each TrieNode is instantiated as a distinct object using native hash maps for character routing which causes

significant memory overhead. If the benchmarking were implemented in lower level languages specifically with direct memory management and struct packing (C++, Rust, etc.), the space complexity footprint should be reduced.

3. Hardware Cache Locality: The algorithmic time measurement covers the whole execution duration but does not isolate CPU cache locality. Linear arrays are stored in contiguous memory blocks which makes them cache-friendly. Conversely, Trie structures rely on pointer chasing across non contiguous memory locations which can lead to many cache misses. While the $O(M)$ complexity confidently overtakes the hardware penalties, cache mechanics still bring unmeasured microsecond variables into the latency table.

IV. Conclusion

This paper has demonstrated the middle ground between discrete mathematics and software engineering through the optimization of real time text validation. The comparative analysis between a traditional linear array search and a prefix tree (Trie) data structures assures a validation in theoretical algorithmic complexities.

The results of the benchmarking also describes the limitations of the linear data structures in interactive applications such as typing games. Operating within a $O(N \times M)$ time complexity, the linear array does not perform well in latency as the lexicon size increases, taking over 635 milliseconds to validate queries against an 80.000 word dataset. Such processing delays can be disrupting or even considered unusable for high frequency usage systems such as the *Typeracer* or *Monkeytype* typing games where instant text validation is a necessity to maintain synchronization and prevent user input lag.

Conversely, the implementation of the Trie structure improves upon the dataset size factor, bounding the search time to the length of the queried string, $O(M)$. This change stabilized execution latencies down to at least 0.94 milliseconds against a similarly sized 80.000 word dictionary, offering an improvement in speed. However, the space complexity analysis revealed that the Trie structure comes at the expense of significantly more physical memory usage ($O(\Sigma \times N \times M)$) to maintain its pointer dictionaries. For an 80.000 word dictionary, a Trie structure requires 68.657 MB, almost 100 times more memory needed compared to the linear array's mere 0.695 MB but is still negligible within modern computing environment standards.

Overall, mapping strings onto a rooted tree graph proves to be one of the mathematical solutions for text validation problems in software engineering. The Trie structure improves validation responses down to within milliseconds ($O(M)$), ensuring the computational backend of typing applications remains synchronized with user input speeds.

V. Appendix

Attached below is a GitHub gist link containing the two Python programs used to measure the time and space complexity of both linear array search and Trie structure lookup algorithms.
https://gist.github.com/M1m1rr/543a60dc8ce211f6ac66a59bff597ae0

References

[1] Rinaldi. https://informatika.stei.itb.ac.id/~rinaldi.munir/

[2] E. Fredkin, "Trie Memory," Communications of the ACM, vol. 3, no. 9, pp. 490-499, Sep. 1960.

[3] R. Sedgewick and K. Wayne, Algorithms, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2011.

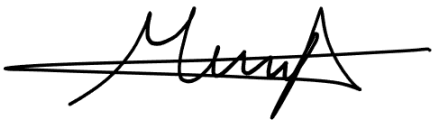
[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA: MIT Press, 2022.

[5] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge, UK: Cambridge University Press, 2002.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 16 Juni 2026



Mirza Aryasatya Akmal
13525131