Application of Tree-Structured Aho-Corasick Algorithm for Offensive Word Detection in Game Chat Systems

Arla Salsabila - 13525086
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: arlas.81194@gmail.com , 13525086@std.stei.itb.ac.id

***Abstract*—In online multiplayer games, real-time chat systems are important for player interaction. However, players often use offensive words, which can corrupt children's mind who are playing the same game. To fix this, game developers must use automated text filters. The problem is that the standard string-matching algorithms are very slow when they have to check a lot of blacklisted words in long chat messages in real time. Aho-Corasick algorithm can solve this issue. This algorithm can find all offensive words in just a single pass by building a tree-structured keyword. This means the processing time depends only on the length of the message, not the size of the dictionary.**

Keywords—Aho-Corasick Algorithm, Tree, Trie, Chat Filtering, Text Moderation.

I. Introduction

Communication is a big part of our everyday lives, including in online multiplayer games. Players use the in-game chat to plan strategies or talk with their friends. Unfortunately, chat rooms are prone to toxic behavior, such as insults, hate speech, and offensive words. To keep the community healthy, game servers need a filtering system that can automatically censor these words before the message is sent to the other players.

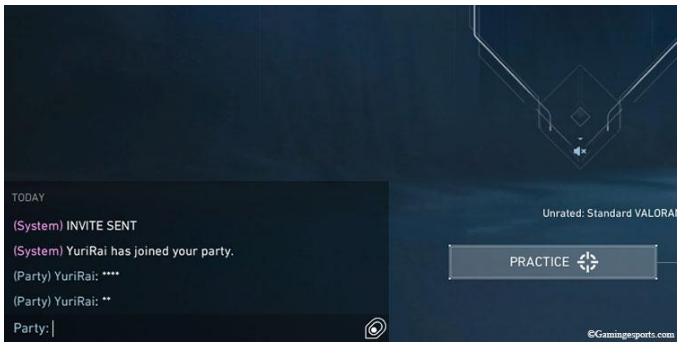

Fig 1.1 Valorant In-Game Chat
(Taken from gamingesports.com)

This is a multi-pattern string matching problem. The game server has a dictionary containing thousands of offensive words that should be blacklisted. Every time a player types something, the system must check if any of those offensive words are inside the message.

If we use a naive approach, the we have to loop through every single word that is blacklisted. If the dictionary has more than ten thousand words and thousands of players are chatting at the same time, then this approach will significantly reduces server efficiency.

That is why we are using Aho-Corasick algorithm which is using concepts like trees. It constructs a data structure that can search for all keywords simultaneously. This means that the dictionary size no longer affects the search speed at all. In this paper, we will implement Aho-Corasick algorithm for game chat filtering using Python.

II. Theoretical Background

A. Tree Theory

In discrete mathematics, a tree is defined as an undirected graph that is connected and does not contain circuits or cycles. In string-searching applications, the type of tree architecture utilized is a rooted tree. A rooted tree is a tree in which one specific vertex is treated as the root, and all edges are assigned directions pointing away from this root, essentially transforming it into a directed graph. Within this hierarchical framework, several key terminologies are defined to describe the relationships between nodes:

- Parent: A node that is located exactly one level directly above another connected node.
- Child: A direct descendant node located exactly one level below its parent node.
- Leaf: Terminal nodes within the tree that do not have any children (their out-degree is equal to zero).
- Ancestor: All nodes along the path from a specific node back up to the root.
- Descendants: All nodes reachable along paths moving down from a specific node.

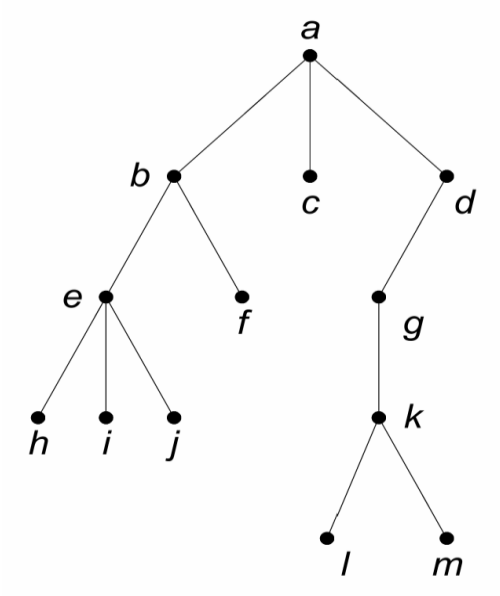

Fig 2.1 Rooted Tree
(Taken from Rinaldi Munir Webpage)

Trie data structure or commonly known as Prefix Tree is designed to store a set of keys represented as strings, making it highly efficient retrieval and storage across large datasets. Unlike a regular search tree that stores a full word in a single node, a Trie distributes strings into a series of interconnected nodes based on prefixes.

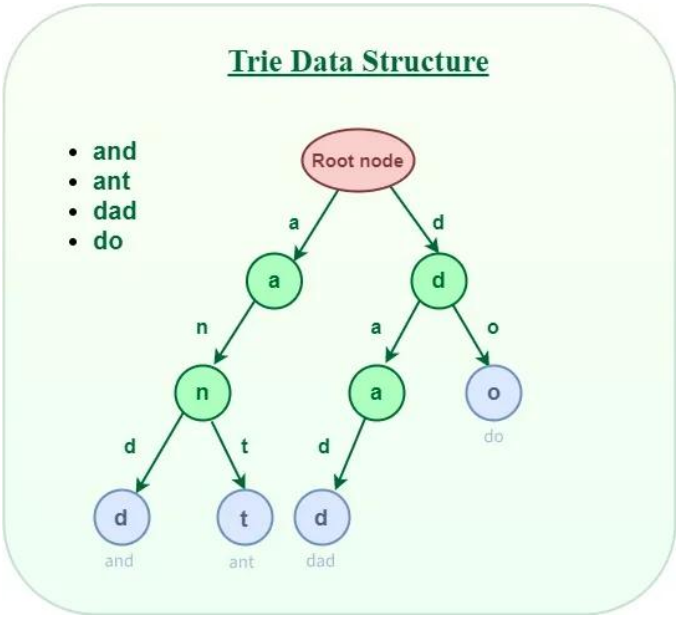

Fig 2.2 Trie Data Structure
(Taken from geeksforgeeks.org)

A Trie for an automated game chat filtering system includes:

- Root Node: Represents the starting point of any text evaluation or an empty string.
- Edges: Each edge is labeled with a single character or alphabet letter taken from the blacklist dictionary.
- Leaf/Output Nodes: Indicate that the sequence of characters read from the root up to that specific node successfully forms a complete offensive keyword from the blocklist.

By implementing this tree-based Trie structure, storing thousands of offensive words becomes significantly more memory-efficient. Keywords that share identical prefixes do not require duplicate nodes; instead, they naturally share the exact same path within the tree structure.

B. Aho-Corasick Algorithm

The Aho-Corasick algorithm was proposed by Alfred Aho and Margaret Corasick in 1975. This algorithm is a string-searching algorithm designed to locate multiple patterns simultaneously. It operates by upgrading a standard Prefix Tree (Trie) data structure into a Deterministic Finite Automaton (DFA). This structural enhancement allows the system to evaluate all target keywords within an incoming text stream in a single-pass scan, completely eliminating the need for backtracking.

To achieve linear-time string matching, the Aho-Corasick automaton relies on three core functions:

- Go to Function: Controls the state transition from the current node to a child node based on the next character read from the text. If the character matches a labeled edge in the tree, the transition succeeds and moves down the hierarchy.
- Failure Function: When the Go to function fails due to a character mismatch, the Failure function shifts the search pointer to an alternate node instead of resetting to the root. This destination node represents the longest proper suffix of the string processed so far that also constitutes a valid prefix elsewhere within the Trie.
- Output Function: Acts as the detection register. Every node in the automaton is associated with a lookup list of matched keywords. When the search pointer reaches or transitions through a node with a valid output status, the system instantly flags that the corresponding keyword has been successfully detected in the text.

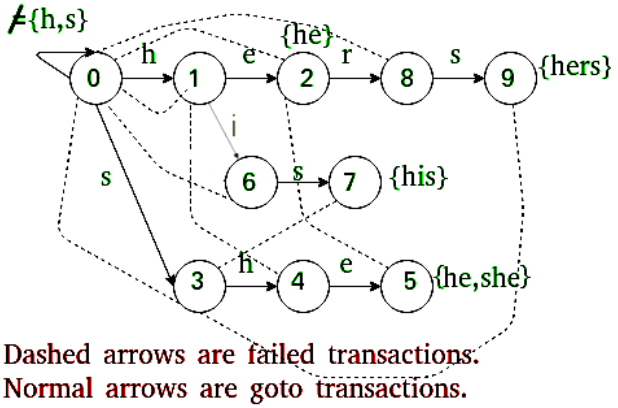

Fig 2.3 Aho-Corasick Structure for Keywords “he”, “she”, “hers”, and “his”
(Taken from geeksforgeeks.org)

The design of this multi-pattern string-matching machine is illustrated in Figure 2.3. Within this automaton, several core components regulate the backtracking-free text search process:

- States (Circles): Represent the current matching positions, beginning from state 0 as the root.
- Solid Arrows (Goto Function): Represent forward transitions following the standard Trie structure when characters from the input text match the pattern.
- Dashed Arrows (Failure Function): Represent automatic fallback links when a character mismatch occurs. These links redirect the search to a state representing the longest proper suffix that is a valid prefix of another pattern, preventing the search from resetting to the root (state 0).
- Labels Next to States (Output Function): Indicate that specific target keywords (such as "he" or "she") have been fully and successfully identified at that particular state.

To illustrate how the automaton processes text, consider the keywords "he", "she", "his", and "hers" as shown in Figure 2.3, with the input text "ushers". The search proceeds character by character as follows:

1. Read 'u' → no matching transition from root → stay at root (state 0)
2. Read 's' → transition to state 3
3. Read 'h' → transition to state 4
4. Read 'e' → transition to state 5 → Output: "she", "he" detected
5. Read 'r' → transition to state 8
6. Read 's' → transition to state 9 → Output: "hers" detected

Without failure links, a mismatch would force the pointer back to the root, causing characters to be re-read. The failure function eliminates this by redirecting the pointer to the longest valid suffix. For example, after reading "she" at state 5, the failure link points to state 2 (representing "he"), allowing "he" to be reported simultaneously without any additional scan.

C. *Algorithm Time Complexity*

Based on the time complexity theory, the performance of an algorithm is evaluated by how its execution time ($T(n)$) grows as the input size (n) increases toward infinity. To establish the upper bound of this growth rate, Big-O Notation is formally applied. A function $T(n)$ is defined as $O(f(n))$ if there exist positive constants C and n_0 such that:

$$T(n) \leq C \cdot f(n) \text{ for all } n \geq n_0$$

The Naive approach checks each keyword in the blacklist one by one against the entire message. Let n be the length of the input text, k be the number of keywords in the blacklist, and m be the average length of each keyword. For each of the k keywords, the algorithm performs a full scan of the text of length n, with each character comparison taking $O(m)$ time in the worst case. Therefore, the total time complexity of the Naive approach is:

$$T(n) = O(n \times m \times k)$$

This means that as the blacklist grows (larger k) or messages become longer (larger n), the processing time increases multiplicatively, making this approach poorly suited for large-scale game chat systems.

For the Aho-Corasick algorithm, the asymptotic time complexity is evaluated across two distinct operational phases:

- Preprocessing Phase: This phase encompasses building the Trie structure from the keyword dictionary and computing the *go to*, *failure*, and *output*. The time complexity for this construction is $O(m)$, where m represents the total accumulated length (total number of characters) of all keywords stored in the dictionary.
- Matching Phase: This phase evaluates the target text by processing each character sequentially from beginning to end. Utilizing the precomputed *failure* links allows the search pointer to shift between branches deterministically without re-reading text characters (backtracking). The time complexity for this pattern-matching phase is $O(n + z)$, where n is the length of the text stream being scanned and z is the total number of keyword occurrences identified.

By combining both phases, the total time complexity of the Aho-Corasick algorithm is $O(n + m + z)$. Since the preprocessing phase runs only once, the per-message cost is effectively $O(n + z)$, which is independent of k (the size of the blacklist dictionary). This linear time complexity guarantees optimal efficiency for a game chat filtering system, ensuring that text evaluation performance remains highly responsive regardless of how large the blacklist dictionary grows.

III. Implementation

The implementation is divided into two main modules: the Naive (Brute Force) filter and the Aho-Corasick automaton, both written in Python. The driver code handles user input,

runs both algorithms against the same chat message, and prints a side-by-side performance comparison.

A. *Naive (Brute Force) Filter*

```python
def naive_filter(text, blacklist, mask_char="*"):
    censored_text = text
    found_words = []

    for keyword in blacklist:
        keyword_len = len(keyword)
        text_len = len(censored_text)

        i = 0
        while i <= text_len - keyword_len:
            substring = censored_text[i:i + keyword_len]

            if substring.lower() == keyword.lower():
                found_words.append((keyword, i))
                mask = mask_char * keyword_len
                censored_text = (
                    censored_text[:i] + mask + censored_text[i + keyword_len:]
                )
                i += keyword_len
                text_len = len(censored_text)
            else:
                i += 1

    return censored_text, found_words
```

Fig 3.1 Naive (Brute Force) Filter

The *naive_filter(text, blacklist, mask_char="*")* function implements the brute force approach directly. For each keyword in the blacklist, the function slides a window across the text one character at a time. At each position *i*, it extracts the substring censored_*text[i:i + keyword_len]* and compares it against the keyword in a case-insensitive manner using *.lower()*. If a match is found, the word is recorded into found_words along with its starting index, and the corresponding characters in the text are replaced with *mask_char * keyword_len*. The pointer *i* then advances by the keyword length to avoid overlap. Otherwise, it advances by one step.

This approach is straightforward but carries a time complexity of $O(n \times m \times k)$, meaning every additional keyword in the blacklist directly and linearly increases execution time.

B. *Aho-Corasick Automaton*

The Aho-Corasick implementation is encapsulated in the AhoCorasick class, structured across three major phases: Trie construction, Failure Function computation, and the search process.

1. Initialization and Data Structures

```python
class AhoCorasick:
    def __init__(self):
        self.goto = [{}]
        self.fail = [0]
        self.output = [[]]
        self.num_states = 1
```

Fig 3.2 AhoCorasick __init__ Method

In the constructor *__init__*, three core data structures are initialized:

- *self.goto*: a list of dictionaries where each element represents one state and maps a character to its next state (goto transitions).
- *self.fail*: a list of integers storing each state's failure link, i.e., the fallback state when a character mismatch occurs.
- *self.output*: a list of lists storing the keywords that are recognized at each state.

State 0 serves as the root of the automaton, and all structures begin with this single state.

2. Trie Construction

```python
def build_trie(self, blacklist):
    for keyword in blacklist:
        current_state = 0
        normalized_keyword = keyword.lower()

        for char in normalized_keyword:
            if char not in self.goto[current_state]:
                self.goto.append({})
                self.fail.append(0)
                self.output.append([])
                self.goto[current_state][char] = self.num_states
                self.num_states += 1

            current_state = self.goto[current_state][char]

        self.output[current_state].append(keyword)
```

Fig 3.3 AhoCorasixk build_trie Method

The *build_trie(blacklist)* method inserts each keyword from the blacklist into the Trie one by one. For each character in a keyword (normalized to lowercase with *.lower()*), the function checks whether a transition for that character already exists at the current state. If it does not, a new state is created by appending new entries to *self.goto, self.fail,* and *self.output,* and *self.num_states* is incremented. Once all characters of a keyword are processed, the final reached state is marked by storing the keyword into *self.output[current_state]*.

This structure ensures that keywords sharing the same prefix naturally share a single path of nodes, making memory usage significantly more efficient than storing each word independently.

3. Failure Function Construction

```
def build_failure_function(self):
    queue = []

    for char, state in self.goto[0].items():
        self.fail[state] = 0
        queue.append(state)

    head = 0
    while head < len(queue):
        current_state = queue[head]
        head += 1

        for char, next_state in self.goto[current_state].items():
            queue.append(next_state)

            fail_state = self.fail[current_state]
            while fail_state != 0 and char not in self.goto[fail_state]:
                fail_state = self.fail[fail_state]

            if char in self.goto[fail_state] and self.goto[fail_state][char] != next_state:
                self.fail[next_state] = self.goto[fail_state][char]
            else:
                self.fail[next_state] = 0

            self.output[next_state] += self.output[self.fail[next_state]]
```

Fig 3.4 AhoCorasick build_failure_function Method

This is the core of Aho-Corasick's advantage over the Naive approach. The method runs a Breadth-First Search (BFS) using a queue. All first-depth states (direct children of the root) are initialized with a failure link pointing to state 0. For each state processed thereafter, the algorithm traverses the failure link chain of its parent state to find the correct failure destination for each of its children. Once a failure link is determined, the output of that failure state is merged into the current state's output list (*self.output[next_state] += self.output[self.fail[next_state]]*). This merging ensures that all keywords which are valid suffixes of the currently processed path are also detected without any additional scan.

4. Search Process

```
def search(self, text):
    current_state = 0
    matches = []
    normalized_text = text.lower()

    for i, char in enumerate(normalized_text):
        while current_state != 0 and char not in self.goto[current_state]:
            current_state = self.fail[current_state]

        if char in self.goto[current_state]:
            current_state = self.goto[current_state][char]
        else:
            current_state = 0

        for keyword in self.output[current_state]:
            start_index = i - len(keyword) + 1
            matches.append((keyword, start_index))

    return matches
```

Fig 3.5 AhoCorasick search Method

The *search(text)* method performs a single-pass scan over the input text. For each character char at position *i*, the algorithm does the following:

- If there is no transition from current_state for char, follow the failure link (*current_state = self.fail[current_state]*) until a valid transition is found or the root is reached.
- If a transition exists, move to the next state through *self.goto*.
- Check *self.output[current_state]*. If non-empty, all keywords stored there have been matched, and each keyword's starting position is calculated as *i - len(keyword) + 1*.

The failure link mechanism is precisely what eliminates backtracking, keeping the matching time complexity at *O(n + z)* regardless of how large the blacklist is.

5. Text Censoring

```
def filter_text(self, text, mask_char="*"):
    matches = self.search(text)
    text_chars = list(text)

    for keyword, start_index in matches:
        for j in range(start_index, start_index + len(keyword)):
            text_chars[j] = mask_char

    censored_text = "".join(text_chars)
    return censored_text, matches
```

Fig 3.6 AhoCorasick filter_text Method

The *filter_text(text)* method calls *search(text)* to retrieve all matches, then converts the text into a list of individual characters. Every character within the range *[start_index, start_index + len(keyword)]* for each match is replaced with *mask_char*. The resulting censored text is then joined back and returned alongside the list of all detected matches.

C. Driver Code and Benchmarking

The driver code accepts chat input directly from the terminal using *input()*, simulating a real player message. After the input is received:

- The Naive algorithm is executed and its runtime is measured using *time.perf_counter()*, Python's high-resolution timer.
- An AhoCorasick object is and the search is executed, all wrapped within a single time measurement to ensure a fair comparison.
- The censored text, total detected words, and execution time from both approaches are printed side by side.

The blacklist used contains 52 words commonly found in toxic online game chat contexts, large enough to meaningfully demonstrate the scalability difference between the two algorithms.

IV. Discussion

For testing the result of our implementation, we input the text *"you are such a noob and an idiot, totally trash*

gameplay" to our program. The program should have detect "noob", "idiot", and "trash" as offensive words and censor them.

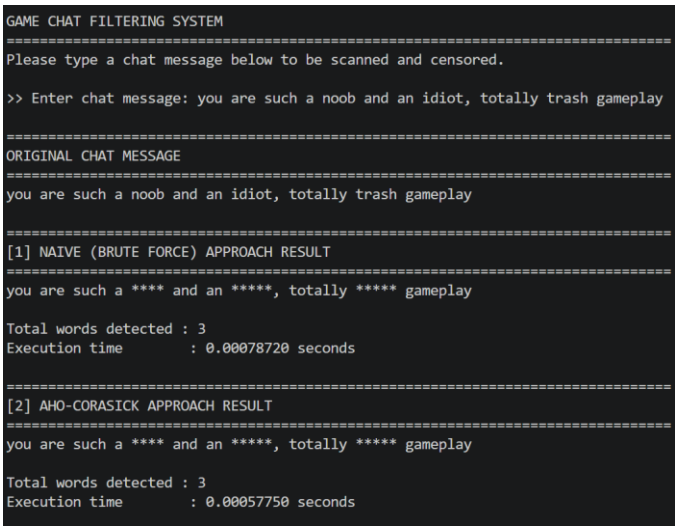

Fig 4.1 Output of The Chat Program

Both algorithms produce identical censored output for the same input, validating the correctness of both implementations. Both algorithms produce *"you are such a **** and an *****, totally ***** gameplay"* with three words detected as offensive, that is "noob", "idiot", and "trash".

The screenshot above shows the execution time of both algorithms on the same input message. While both produce identical output, a measurable time difference is visible. The Aho-Corasick algorithm consistently finishes faster than the Naive approach, even with preprocessing included in the measurement. This result aligns directly with the theoretical complexity gap between $O(n \times m \times k)$ and $O(n + m + z)$.

It is worth noting that in this small-scale test, the absolute time difference may appear narrow due to the relatively short input message and modest blacklist size. The most critical observation is the role of k (blacklist size). In the Naive approach, doubling the number of blacklisted words nearly doubles the execution time per message. In Aho-Corasick, the blacklist size has zero impact on matching speed because the entire keyword dictionary is encoded into the automaton during a one-time preprocessing phase. Once built, the automaton processes any incoming message in a single left-to-right pass regardless of whether the blacklist contains 50 or 50,000 words.

In a production game server environment, such as a competitive multiplayer title handling thousands of concurrent players from across the world, this distinction becomes the deciding factor between a responsive filtering system and one that creates latency bottlenecks.

Despite its superior matching speed, the Aho-Corasick implementation carries several considerations worth acknowledging:

- Memory overhead: The go to, fail, and output structures collectively consume more memory than the Naive approach, which requires no preprocessing state at all. For extremely memory-constrained environments, this trade-off must be evaluated carefully.
- Implementation complexity: The Aho-Corasick code is considerably more complex, particularly the build_failure_function method. This increases the risk of subtle bugs during development and makes the codebase harder to maintain without proper documentation.
- Intentional obfuscation: The current implementation does not handle common evasion techniques such as leet speak (n00b, 1d10t) or deliberate misspellings (stu.pid). A production-grade system would require a text normalization layer applied to the input before it enters the automaton.

Despite these limitations, Aho-Corasick remains as the practically superior choice for large-scale game chat filtering over the Naive approach.

V. Conclusion

This paper presented the application of the Aho-Corasick algorithm as an efficient solution for multi-pattern string matching in game chat filtering systems. By building upon tree-based data structures, the algorithm is capable of detecting all offensive keywords within an incoming chat message in a single pass, completely eliminating the need for backtracking.

The implementation demonstrated that both the Naive and Aho-Corasick approaches produce identical censored output, confirming the correctness of both methods. However, the two approaches differ fundamentally in how they scale. The Naive approach operates at $O(n \times m \times k)$, meaning its execution time grows multiplicatively as the blacklist expands or messages grow longer. The Aho-Corasick algorithm, by contrast, achieves an overall complexity of $O(n + m + z)$, where the blacklist size k has no influence on per-message matching speed whatsoever. The keyword dictionary is encoded once during preprocessing, and every subsequent message is evaluated against the full blacklist in linear time relative only to the message length.

These findings confirm that for real-world game chat systems, where blacklists can contain tens of thousands of words across multiple languages and servers must handle thousands of concurrent players in real time, the Aho-Corasick algorithm is the algorithmically and practically superior choice. The Naive approach, while simpler to implement and understand, is fundamentally unsuitable for production-scale deployment due to its multiplicative time complexity.

Future work could extend this implementation in several directions. First, incorporating a text normalization layer before the automaton to handle intentional obfuscation such as leet speak and deliberate misspellings would significantly improve detection coverage. Second, supporting dynamic blacklist updates without requiring a full automaton rebuild would be valuable for live game environments where new offensive terms emerge continuously. Third, extending the system to

handle multilingual blacklists would broaden its applicability across global player bases.

ACKNOWLEDGMENT

The author would like to thank God for all the guidance throughout the process of learning the materials needed and writing this paper. The author would also like to thank the lecturer of ITB Discrete Mathematics IF 1220, Mr. Rinaldi Munir for sharing his knowledge and guide the students throughout the learning process in the class. The author also would like to thank her family and friends who have given their support throughout the entire semester.

REFERENCES

[1] Rinaldi Munir, "Pohon (Bag. 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/23-Pohon-Bag1-2026.pdf, accessed on June 19, 2026 at 19.00.

[2] Rinaldi Munir, "Pohon (Bag. 2)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf, accessed on June 19, 2026 at 19.00.

[3] Rinaldi Munir, "Kompleksitas Algoritma (Bagian 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf, accessed on June 19, 2026 at 19.00.

[4] Rinaldi Munir, "Kompleksitas Algoritma (Bagian 2)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf, accessed on June 19, 2026 at 19.00.

[5] https://www.geeksforgeeks.org/dsa/trie-insert-and-search/, accessed on June 19, 2026 at 19.00.

[6] https://www.geeksforgeeks.org/dsa/aho-corasick-algorithm-pattern-searching/, accessed on June 19, 2026 at 19.00.

[7] https://cp-algorithms.com/string/aho_corasick.html, accessed on June 19, 2026 at 19.00.

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, 19 Juni 2026

Arla Salsabila
13525086