Analysis of Suffix Trees and Suffix Arrays

Tanush Garg

1 Introduction: Suffix Data Structures

In computer science, Suffix Trees and Suffix Arrays are powerful data structures built upon a single string (or "text"). Their primary purpose is to pre-process a text T of length n to answer complex string queries in an exceptionally fast manner.

What the Algorithm Solves The fundamental problem these structures solve is the **substring query problem**: given a text T and a new pattern P of length m, does P exist as a substring within T?

A naive search (checking every possible starting position in T) takes $O(n \times m)$ time. While algorithms like KMP or Boyer-Moore improve this to $O(n + m)$ for a single search, they do not pre-process the text T.

Suffix trees and arrays adopt a different strategy: they invest in an upfront "preprocessing" cost to build the data structure (e.g., $O(n)$). After this one-time cost, any subsequent substring query for any pattern P can be answered in $O(m)$ (Suffix Tree) or $O(m \log n)$ (Suffix Array) time.

Uses and Applications This model is ideal for applications where the text is large and fixed, while patterns are numerous and unknown ahead of time. This perfectly describes the context of bioinformatics and DNA sequence analysis, as mentioned in our project proposal.

- **Text (T):** A complete genome (e.g., E. coli), which is static.
- **Patterns (P):** Thousands of different DNA motifs, gene sequences, or promoter regions to be searched.

Beyond exact substring matching, these structures can efficiently solve many other problems:

- Finding the longest repeated substring in T.
- Finding the longest common substring between two different texts T_1 and T_2.
- Finding all occurrences of P in T (not just the first).
- Problems related to string matching with mismatches (in more advanced forms).

2 The Algorithm: Structure and Function

2.1 Suffix Array (SA)

A Suffix Array is conceptually the simpler of the two structures.

What it is For a text T of length n (we assume T ends with a special character '$' that is lexicographically smaller than any other character), the Suffix Array 'SA' is an array of integers from 0 to $n - 1$.

It stores the **starting indices** of all suffixes of T in **lexicographical (sorted) order**.

Example Let T = "banana$" (length $n = 7$). The suffixes are:

- 0: banana$
- 1: anana$
- 2: nana$
- 3: ana$
- 4: na$
- 5: a$
- 6: $

When sorted lexicographically:

1. (index 6) $
2. (index 5) a$
3. (index 3) ana$
4. (index 1) anana$
5. (index 0) banana$
6. (index 4) na$
7. (index 2) nana$

The Suffix Array ‘SA‘ is the array of starting indices: SA $= [6, 5, 3, 1, 0, 4, 2]$

How it Works (Search) To find a pattern P (e.g., "ana"), we can perform a binary search on the Suffix Array ‘SA‘.

1. Pick the middle element of ‘SA‘, e.g., SA[3] = 1.
2. Compare P ("ana") with the suffix starting at index 1: $T[1...]$ = "anana$".
3. "ana" is lexicographically smaller than "anana$".
4. We adjust our binary search to the "left" half of ‘SA‘: $[6, 5, 3]$.
5. Pick the middle element, SA[1] = 5.
6. Compare P ("ana") with $T[5...]$ = "a$".
7. "ana" is larger than "a$".
8. We adjust to the "right" half: [3].
9. Pick SA[2] = 3. Compare P ("ana") with $T[3...]$ = "ana$".
10. "ana" is a prefix of "ana$". A match is found.

All suffixes beginning with "ana" (i.e., "ana$" and "anana$") will appear in a contiguous block in the sorted list, and thus their indices (3 and 1) will be contiguous in the ‘SA‘.

2.2 Suffix Tree (ST)

A Suffix Tree is a more complex but faster data structure.

What it is A Suffix Tree is a **compressed trie** (also known as a "patricia trie") containing all n suffixes of the text T.

- It has a root node and n leaves.
- Each leaf corresponds to one suffix of T.
- Each internal node (except the root) has at least two children.
- Each edge is labeled with a non-empty substring of T.
- "Compressed" means that paths with no branches are compressed into a single edge. (e.g., a path ‘r -> o -> o -> t‘ would become a single edge ‘r -> (root)‘ labeled "oot").
- The concatenation of edge labels on the path from the root to leaf i spells out the suffix $T[i...n-1]$.

How it Works (Search) To find a pattern P of length m, we "thread" P down from the root of the tree.

1. Start at the root.
2. Find an edge whose label starts with the first character of P.
3. If no such edge exists, P is not in T.
4. If an edge is found, "consume" the matching portion of P and the edge label.
5. **Case 1: Pattern is consumed.** If we consume all of P (and end up either in the middle of an edge or at a node), then P exists in T.
6. **Case 2: Edge is consumed.** If we consume an entire edge label and still have part of P remaining, we continue the search from the child node that edge points to.
7. **Case 3: Mismatch.** If the next character in P does not match the next character on the edge label, P is not in T.

This process is a simple walk from the root, and each character in P is examined exactly once.

3 Time and Space Complexity Analysis

Let n be the length of the text T and m be the length of the pattern P.

3.1 Suffix Tree

- **Preprocessing (Construction):** Advanced algorithms like Ukkonen's (1995) or McCreight's (1976) can construct a Suffix Tree in $O(n)$ time. Simpler algorithms, like inserting each suffix into a trie, take $O(n^2)$ time.
- **Search (Query):** As described above, searching for P involves a single walk from the root. At each node, finding the correct outgoing edge can be done in $O(1)$ time (if the alphabet size is constant and a lookup table is used) or $O(\log k)$ (if edges are sorted, where k is alphabet size). In the worst case, we examine each of P's m characters once. Search time is $O(m)$.

- **Space Complexity:** A Suffix Tree has n leaves and at most $n-1$ internal nodes, for a total of $O(n)$ nodes. Each node must store pointers to its children. This leads to an $O(n)$ space complexity.

  Note: While asymptotically $O(n)$, the constant factor for a Suffix Tree is very large. Each node object requires significant overhead (pointers, indices), making Suffix Trees extremely memory-intensive in practice, as noted in the project brief.

3.2 Suffix Array

- **Preprocessing (Construction):** Naive construction (generating all suffixes and sorting) takes $O(n^2 \log n)$ time. More advanced "prefix doubling" or Manber-Myers algorithms construct the array in $O(n \log n)$ time. State-of-the-art algorithms (like DC3/Skew or SA-IS) achieve $O(n)$ time.

- **Search (Query):** A standard binary search on the ‘SA‘ requires $O(\log n)$ comparisons. Each comparison may need to compare up to m characters. Search time is $O(m \log n)$.

  This can be improved to $O(m + \log n)$ by using an auxiliary **LCP (Longest Common Prefix) array**, which is beyond this initial scope but is a standard optimization.

- **Space Complexity:** The Suffix Array itself is just an array of n integers. Space is $O(n)$.

  Note: The constant factor here is much smaller than for a Suffix Tree. It is simply the size of n integers (e.g., $n \times 8$ bytes for 64-bit integers), making it far more practical for large texts like genomes.

4 Implementation Details: Algorithms Behind the Structures

This section presents how the Suffix Array and Suffix Tree can be constructed in practice. We first examine the naive (brute-force) methods to build intuition, and then the optimized Manber–Myers and Ukkonen algorithms that achieve $O(n \log n)$ and $O(n)$ construction times, respectively.

4.1 Naive Construction of the Suffix Array

High-Level Idea. The brute-force way to build a Suffix Array is conceptually simple:

Step 1. Generate all n suffixes of T.

Step 2. Sort them lexicographically (like sorting n strings).

Step 3. Record the starting indices of the sorted suffixes.

Example. For $T =$ `banana$`, the suffixes are:

$$[\texttt{banana\$}, \texttt{anana\$}, \texttt{nana\$}, \texttt{ana\$}, \texttt{na\$}, \texttt{a\$}, \texttt{\$}]$$

After sorting them lexicographically, we get:

$$[\texttt{\$}, \texttt{a\$}, \texttt{ana\$}, \texttt{anana\$}, \texttt{banana\$}, \texttt{na\$}, \texttt{nana\$}]$$

Thus, the Suffix Array is:

$$SA = [6, 5, 3, 1, 0, 4, 2]$$

Complexity.

- Generating suffixes: $O(n^2)$ (each suffix may be length $O(n)$)
- Sorting n strings: $O(n \log n)$ comparisons, each $O(n)$ time

Total complexity:

$$O(n^2 \log n)$$

which is infeasible for long texts (e.g., genome sequences).

4.2 Optimized Construction: Manber–Myers Algorithm

Overview. The **Manber–Myers algorithm** (1993) constructs the Suffix Array in $O(n \log n)$ time using the technique of **prefix doubling**. Unlike the naive method that compares entire suffix strings (costing $O(n^2 \log n)$), it works by reusing previously computed information about shorter prefixes.

The core insight is:

> If two suffixes differ within their first 2^{k-1} characters, their order is already known. Otherwise, we can determine their order by comparing the next 2^{k-1} characters, whose relative order we also already know from the previous iteration.

Algorithm Intuition.

Step 1. Initialization: Assign each character in T a rank corresponding to its lexicographic order. This gives the correct ordering of all suffixes by their first character.

Step 2. Prefix Doubling: In iteration k, we sort suffixes based on their first 2^k characters by forming pairs $(\text{rank}[i], \text{rank}[i + 2^{k-1}])$ for each index i. These pairs encode the rank of the first half and the rank of the second half of the prefix. Sorting by these integer pairs is equivalent to sorting by 2^k characters.

Step 3. Rank Update: After sorting, assign new ranks to each suffix:

- If a suffix has the same pair as the previous suffix in sorted order, it receives the same rank.
- Otherwise, its rank increases by one.

The updated rank array now represents the order of suffixes based on 2^k-length prefixes.

Step 4. Doubling: Continue doubling ($k \leftarrow 2k$) until all ranks are distinct or $2^k \geq n$, meaning all suffixes are fully ordered.

Why It Works. Each iteration doubles the prefix length over which suffixes are correctly ordered:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow \ldots$$

After $\lceil \log_2 n \rceil$ iterations, the prefixes are at least n characters long, which implies that the suffixes are fully sorted. Formally, sorting by the pair $(\text{rank}[i], \text{rank}[i + 2^{k-1}])$ ensures lexicographic order for the first 2^k characters, since:

- The first element compares the first 2^{k-1} characters.
- The second element breaks ties using the next 2^{k-1} characters.

Dry Run Example. Let $T =$ `banana$` ($n = 7$).

Step 0: Initial Ranking (1 Character). Assign ranks to characters:

$$\$ \to 0, \quad a \to 1, \quad b \to 2, \quad n \to 3$$

$$\text{rank} = [2, 1, 3, 1, 3, 1, 0]$$

Iteration 1 ($2^0 = 1$): Sort by (rank[i], rank[i+1]) Each suffix is represented by a pair of ranks corresponding to its first two characters:

$$[(2, 1), (1, 3), (3, 1), (1, 3), (3, 1), (1, 0), (0, -1)]$$

Sorting by these pairs gives the order of suffixes by their first two characters:

$$SA = [6, 5, 3, 1, 4, 2, 0]$$

New ranks are assigned based on sorted order:

$$\text{rank} = [6, 2, 5, 3, 4, 1, 0]$$

Interpretation:

- The smallest suffix starts at index 6 (`$`).
- The next smallest starts at index 5 (`a$`), and so on.

Iteration 2 ($2^1 = 2$): Sort by (rank[i], rank[i+2])

$$[(6, 5), (2, 3), (5, 4), (3, 1), (4, 0), (1, -1), (0, -1)]$$

After sorting:

$$SA = [6, 5, 3, 1, 0, 4, 2]$$

Now each suffix is correctly ordered by its first 4 characters.

Verification. Check the sorted suffixes in lexicographic order:

1. $
2. a$
3. ana$
4. anana$
5. banana$
6. na$
7. nana$

This matches the true lexicographic order, confirming correctness.

Why Ranks Capture Lexicographic Order. At each step, the algorithm effectively treats each rank as a compressed summary of a substring's prefix. If two suffixes share the same rank for the first k characters, they are tied until the next comparison window ($i + 2^k$) breaks the tie. This works because:

$$\text{Lexicographic order of strings} \iff \text{Lexicographic order of their rank tuples.}$$

Complexity Analysis.

- Each iteration sorts n integer pairs in $O(n \log n)$ or $O(n)$ time (if radix sort is used).
- There are $O(\log n)$ iterations.
- **Total:** $O(n \log n)$ time and $O(n)$ space.

Advantages.

- Deterministic and simple to verify step by step.
- Reuses previously computed rank information.
- Much faster than the naive $O(n^2 \log n)$ suffix sorting.
- Widely implemented in genome sequence analysis, text indexing, and pattern matching.

```
FUNCTION BuildSuffixArray(T):
    n = length(T)
    SA = [0, 1, 2, ..., n-1]
    rank = [ord(T[i]) for i in range(n)]
    k = 1

    WHILE k < n:
        // Sort suffix indices by first 2^k characters
        SA.sort(key=lambda i: (rank[i],
                    rank[i+k] if i+k < n else -1))

        // Assign new ranks based on sorted pairs
        new_rank = [0] * n
        new_rank[SA[0]] = 0

        FOR i IN range(1, n):
            prev, curr = SA[i-1], SA[i]
            prev_pair = (rank[prev],
                         rank[prev+k] if prev+k < n else -1)
            curr_pair = (rank[curr],
                         rank[curr+k] if curr+k < n else -1)

            IF curr_pair != prev_pair:
                new_rank[curr] = new_rank[prev] + 1
            ELSE:
                new_rank[curr] = new_rank[prev]

        rank = new_rank
        k *= 2

    RETURN SA
```

Listing 1: Manber–Myers Suffix Array Construction Algorithm

Key Insight.

> Each suffix's rank acts as a "summary" of its prefix. By combining two existing ranks, the algorithm effectively compares twice as many characters without re-checking the text. After $\log_2 n$ rounds, the ranks encode the full suffix order.

4.3 Naive Construction of the Suffix Tree

High-Level Idea. The naive Suffix Tree construction algorithm directly inserts each of the n suffixes of T into a standard trie. For each suffix $T[i..n-1]$, we traverse character by character from the root, creating new nodes as needed.

Steps.

Step 1. Initialize an empty root node.

Step 2. For each i from 0 to $n-1$:

- Start at the root.
- For each character in $T[i..n-1]$, follow existing edges if they match; otherwise, create a new edge.

Example. For $T =$ `banana$`, the tree would have 7 leaf paths:

`banana$`, `anana$`, `nana$`, `ana$`, `na$`, `a$`, `$`

Each path is distinct, but this leads to $O(n^2)$ total nodes and edges.

Complexity. Inserting all n suffixes of length up to n gives $O(n^2)$ time and space.

4.4 Optimized Construction: Ukkonen's Algorithm

Overview. Ukkonen's algorithm (1995) builds the Suffix Tree in linear time, $O(n)$, by extending the tree incrementally one character at a time, maintaining an implicit tree at each phase. It uses key optimizations: **active points** and **suffix links**.

Intuition.

- Each phase adds one new character $T[i]$.
- Instead of reinserting all suffixes, the algorithm maintains the position (the *active point*) from which new extensions should start.
- Suffix links connect internal nodes with similar suffix contexts, avoiding re-traversal from the root.

High-Level Steps.

Step 1. Initialize an empty root.

Step 2. For each character $T[i]$:

- Perform extensions for all active suffixes.
- Split edges when needed to create new internal nodes.
- Update suffix links to optimize the next phase.

Dry Run (Conceptual). Let $T =$ `aba$`.

Phase 1: Add `a`: Tree has one path `a$`.

Phase 2: Add `b`: Paths `a$`, `b$`.

Phase 3: Add `a`: Creates new branches for `a$`, `ba$`, and `aba$`.

Phase 4: Add `$`: Terminates all suffixes, completing the explicit tree.

Each phase modifies only part of the tree due to the active point.

Complexity.

Time: $O(n)$ Space: $O(n)$

Summary Comparison.

Algorithm	Time Complexity	Practical Difficulty
Naive Suffix Array	$O(n^2 \log n)$	Very Easy
Manber–Myers SA	$O(n \log n)$	Moderate
Naive Suffix Tree	$O(n^2)$	Easy
Ukkonen ST	$O(n)$	Very High

5 Suffix Trees vs. Arrays: Trade-offs and Use Cases

Given that both structures solve the substring query problem and can be built in $O(n)$ time, the need for two different structures arises from a classic computer science trade-off: **memory vs. speed**.

In short, the Suffix Tree is theoretically faster for queries but uses a large amount of memory, while the Suffix Array is more memory-efficient but theoretically slower for queries.

Suffix Tree (ST) Use Cases

- **Pro (Speed):** The primary advantage is its $O(m)$ query time, which is independent of the text length n. For applications with a massive text and an extremely high volume of short queries, this constant-time-per-character lookup is the fastest possible.
- **Con (Memory):** The $O(n)$ space complexity has a very large constant factor. Each node in the tree requires multiple pointers (to children, parent, suffix links) and indices, leading to high memory overhead. As noted in the project brief, this makes them impractical for very large texts like full genomes, which can be gigabytes in size.
- **Use Case:** Applications where query speed is the absolute bottleneck and available memory is not a constraint. Also, some advanced string problems (like Longest Common Substring of two strings) have a slightly more intuitive $O(n)$ solution using a Generalized Suffix Tree.

Suffix Array (SA) Use Cases

- **Pro (Memory):** This is the Suffix Array's main advantage. Its $O(n)$ space complexity has a very small constant factor—it is simply an array of n integers. This is significantly smaller (often 10x-50x smaller) than its equivalent Suffix Tree.
- **Pro (Simplicity):** While $O(n)$ construction is complex, the data structure itself is just an array, which is simpler to store, serialize, or pass.

- **Con (Speed):** The standard search time is $O(m \log n)$. The $\log n$ factor, while small, makes it technically slower than a Suffix Tree.
- **Use Case:** This is the **de facto standard** for most practical applications involving large texts, especially in bioinformatics. The memory savings are essential for handling genome-scale data, and the $O(m \log n)$ search is almost always "fast enough".

The "Best of Both Worlds": Enhanced Suffix Array (ESA) The trade-off is largely resolved by using a Suffix Array in conjunction with an **LCP (Longest Common Prefix) array**.

- The LCP array is *also* $O(n)$ in space (one integer per suffix) and can be constructed in $O(n)$ time (e.g., using Kasai's algorithm after the SA is built).
- An "Enhanced Suffix Array" (SA + LCP) can be used to simulate all operations of a Suffix Tree, including $O(m)$ **search**, with the same $O(n)$ space-efficiency of the Suffix Array.
- **Conclusion:** Because the Enhanced Suffix Array provides the $O(m)$ query time of the Suffix Tree and the small memory footprint of the Suffix Array, it has become the preferred data structure in almost all modern implementations.

6 Proof of Correctness

6.1 Preliminaries

Let T be a string of length n, and assume T is terminated by a unique character \$ that is lexicographically smaller than any other symbol. This ensures all suffixes are distinct and no suffix is a prefix of another.

Let $S_i = T[i \dots n-1]$ denote the suffix of T starting at index i.

6.2 Correctness of Suffix Tree Search

Claim The Suffix Tree search procedure returns `true` if and only if the pattern P occurs as a substring of T.

Proof We prove correctness by induction on the length $m = |P|$.

Base Case ($m = 0$): The empty string is a substring of every string. The algorithm immediately returns `true`. Correct.

Inductive Step: Assume the algorithm is correct for all patterns of length $< m$. Let $|P| = m > 0$. From the root, the algorithm selects the outgoing edge whose first character matches $P[0]$:

- If no such edge exists, no suffix of T begins with $P[0]$, so P cannot occur in T. Returning `false` is correct.

Let the matching edge be labeled with substring $L = T[a \dots b]$.
The algorithm compares P and L character-by-character:

- **Mismatch Case:** Suppose at position j we have $P[j] \neq L[j]$. Since all strings on this edge begin with L, no suffix can begin with P. Returning `false` is correct.
- **Pattern Fully Consumed ($m \leq |L|$):** All characters of P match the first m characters of L. Since L corresponds to a prefix of a suffix of T, P occurs in T. Returning `true` is correct.
- **Edge Fully Consumed ($m > |L|$):** We have $P[0 \dots |L|-1] = L$, and the remaining pattern is $P' = P[|L| \dots m-1]$. The algorithm recurses into the child node. Since the child subtree contains exactly all suffixes beginning with L, P occurs in T if and only if P' occurs in some suffix. By the inductive hypothesis, the recursive result is correct.

Thus by induction, the Suffix Tree search procedure is correct.

6.3 Correctness of Suffix Array Search

Let SA be the array of suffix indices sorted lexicographically:

$$S_{\text{SA}[0]} < S_{\text{SA}[1]} < \cdots < S_{\text{SA}[n-1]}.$$

Key Lemma (Contiguity) All suffixes beginning with P appear in a single contiguous range in SA.

Proof: Suppose $S_{\text{SA}[i]}$ and $S_{\text{SA}[k]}$ both begin with P and $i < j < k$. If $S_{\text{SA}[j]}$ did not begin with P, then lexicographically it must be either $< P$ or $> P$, contradicting the fact that it lies between two strings with prefix P. Thus the set is contiguous.

Binary Search Correctness Define a predicate:

$$F(k) = (S_{\text{SA}[k]} < P)$$

under lexicographic order. $F(k)$ is monotone: if true at k, it is true for all $j < k$.

Binary search on F finds the smallest index L such that $S_{\text{SA}[L]} \geq P$. There are two cases:

- If no suffix starts with P, then either $L = n$ or $S_{\text{SA}[L]}$ does not begin with P. The algorithm correctly returns `false`.
- If $S_{\text{SA}[L]}$ begins with P, then by contiguity, all matching suffixes lie in a block $[L, R]$. The algorithm correctly returns `true`.

Thus the Suffix Array binary search procedure is correct.

7 Pseudocode (Search Algorithms)

Construction algorithms are omitted due to their high complexity. The search algorithms are more straightforward.

```
/* Node: A node in the suffix tree.
node.children: A map or list of edges (e.g., {'a': edge1, 'c': edge2...})
edge.label: The substring label on that edge (e.g., "nana")
edge.child: The node that edge points to.
*/
FUNCTION SearchST(node, pattern):
  IF pattern is empty:
    RETURN true  // Pattern was fully matched

  // Find the edge corresponding to the first char of the pattern
  first_char = pattern[0]
  edge = node.children.find_edge_starting_with(first_char)

  IF edge is null:
    RETURN false // No match

  // --- Match Found, Check Edge ---
  edge_label = edge.label
  pattern_len = length(pattern)
  label_len = length(edge_label)

  FOR i FROM 0 TO min(pattern_len, label_len):
```

```
    IF pattern[i] != edge_label[i]:
      RETURN false // Mismatch on the edge

  // --- End of loop, no mismatch so far ---
  IF pattern_len <= label_len:
    // Case 1: Pattern is consumed, ending on this edge
    // e.g., pattern="an", edge_label="ana"
    RETURN true
  ELSE:
    // Case 2: Edge is consumed, pattern remains
    // e.g., pattern="anana", edge_label="ana"
    remaining_pattern = pattern[label_len ...]
    RETURN SearchST(edge.child, remaining_pattern)
```

Listing 2: Pseudocode for Suffix Tree Search

```
/*
T: The full text string
SA: The Suffix Array (array of integers)
P: The pattern string to find
*/
FUNCTION SearchSA(T, SA, P):
  n = length(SA)
  low = 0
  high = n - 1
  m = length(P)

  WHILE low <= high:
    mid = (low + high) / 2

    // Get the suffix from the text
    suffix_start_index = SA[mid]

    // Compare P to the suffix T[suffix_start_index...]
    // We only need to compare up to m characters
    comparison_result = compare(
        P,
        T[suffix_start_index : suffix_start_index + m]
    )

    IF comparison_result == 0:
      // P is a prefix of this suffix
      RETURN true // Match found
    ELSE IF comparison_result < 0:
      // P is lexicographically smaller
      high = mid - 1
    ELSE:
      // P is lexicographically larger
      low = mid + 1

  // Loop finished without finding a match
  RETURN false
```

Listing 3: Pseudocode for Suffix Array Binary Search

8 Dry Run on Biological Sequence

This section demonstrates how a **Suffix Array** performs pattern search on a biological sequence. We will trace the algorithm step by step using the same example sequence and restriction site from the Boyer–Moore example.

8.1 Example: Finding EcoRI Restriction Site with a Suffix Array

- **Pattern P:** `GAATTC` ($m = 6$)
- **Text T:** `ACGTACGGATGCGAATTCAGTACG$` ($n = 25$)

Note: The terminator symbol `$` ensures that all suffixes are distinct and lexicographically smaller than any other suffix.

8.1.1 Preprocessing (Suffix Array Construction)

We first generate all suffixes of T and then sort them lexicographically. The resulting sorted order determines the Suffix Array (SA).

1. Generate All Suffixes

0: `ACGTACGGATGCGAATTCAGTACG$`
1: `CGTACGGATGCGAATTCAGTACG$`
2: `GTACGGATGCGAATTCAGTACG$`
3: `TACGGATGCGAATTCAGTACG$`
4: `ACGGATGCGAATTCAGTACG$`
5: `CGGATGCGAATTCAGTACG$`
6: `GGATGCGAATTCAGTACG$`
7: `GATGCGAATTCAGTACG$`
8: `ATGCGAATTCAGTACG$`
9: `TGCGAATTCAGTACG$`
10: `GCGAATTCAGTACG$`
11: `CGAATTCAGTACG$`
12: `GAATTCAGTACG$` ← (pattern prefix)
13: `AATTCAGTACG$`
14: `ATTCAGTACG$`
15: `TTCAGTACG$`
16: `TCAGTACG$`
17: `CAGTACG$`
18: `AGTACG$`
19: `GTACG$`
20: `TACG$`
21: `ACG$`
22: `CG$`
23: `G$`
24: `$`

2. Sort Suffixes Lexicographically

24: $
21: ACG$
4: ACGGATGCGAATTCAGTACG$
0: ACGTACGGATGCGAATTCAGTACG$
18: AGTACG$
13: AATTCAGTACG$
14: ATTCAGTACG$
8: ATGCGAATTCAGTACG$
17: CAGTACG$
22: CG$
11: CGAATTCAGTACG$
5: CGGATGCGAATTCAGTACG$
1: CGTACGGATGCGAATTCAGTACG$
12: GAATTCAGTACG$ ← (match)
7: GATGCGAATTCAGTACG$
10: GCGAATTCAGTACG$
6: GGATGCGAATTCAGTACG$
23: G$
19: GTACG$
2: GTACGGATGCGAATTCAGTACG$
16: TCAGTACG$
20: TACG$
3: TACGGATGCGAATTCAGTACG$
9: TGCGAATTCAGTACG$
15: TTCAGTACG$

3. Final Suffix Array (SA)

$$SA = [24, 21, 4, 0, 18, 13, 14, 8, 17, 22, 11, 5, 1, 12, 7, 10, 6, 23, 19, 2, 16, 20, 3, 9, 15]$$

8.1.2 Search Process (Binary Search)

We perform binary search on SA to locate the pattern $P =$ `GAATTC`.

Step 1. Initial State: `low = 0`, `high = 24`
⇒ `mid = 12`, `SA[12] = 1`
Compare `"GAATTC"` with $T[1..] =$ `CGTACGGATGCGAATTCAGTACG$`
Result: `G > C` ⇒ Pattern is lexicographically larger.
Action: `low = 13`.

Step 2. State: `low = 13`, `high = 24`
⇒ `mid = 18`, `SA[18] = 19`
Compare `"GAATTC"` with $T[19..] =$ `GTACG$`
Result: `A < T` ⇒ Pattern is smaller.
Action: `high = 17`.

Step 3. State: `low = 13`, `high = 17`
⇒ `mid = 15`, `SA[15] = 10`
Compare `"GAATTC"` with $T[10..] =$ `GCGAATTCAGTACG$`
Result: `A < C` ⇒ Pattern is smaller.
Action: `high = 14`.

Step 4. State: `low = 13`, `high = 14`
⇒ `mid = 13`, `SA[13] = 12`

Compare `"GAATTC"` with $T[12..] =$ `GAATTCAGTACG$`
Result: Exact match found.
Action: `RETURN true`.

8.1.3 Observations

- The binary search performs at most $\lceil \log_2 n \rceil = 5$ iterations.
- Each comparison examines up to $m = 6$ characters, for total $O(m \log n)$ time.
- Only four suffix probes were required in a 25-character text.
- The match occurs at text index 12 (0-based), confirming that `GAATTC` starts at position 13 (1-indexed).

8.2 Example: Finding EcoRI Restriction Site with a Suffix Tree

We now perform the same search using a **Suffix Tree**. Unlike the Suffix Array, which is a sorted list of suffix indices, the Suffix Tree is a compressed trie of all suffixes of T.

- **Pattern P:** `GAATTC` ($m = 6$)
- **Text T:** `ACGTACGGATGCGAATTCAGTACG$` ($n = 25$)

Note: The character `$` denotes the end of the text and ensures that no suffix is a prefix of another.

8.2.1 Preprocessing (Suffix Tree Construction)

The Suffix Tree is a compressed trie representing all suffixes of T. Each path from the root to a leaf spells one suffix $T[i..n-1]$.

8.3 Complete Compressed Suffix Tree for the Example

Let

$$T = \texttt{ACGTACGGATGCGAATTCAGTACG\$}$$

(length $n = 25$). The following is the *complete compressed suffix tree* for T. Each leaf is annotated with the starting index of the corresponding suffix in T (0-based). Edge labels are full substrings from T (no truncation).

```
(ROOT)
  "A" →
       "CG" →
            "$"                          [21]   := "ACG$"
            "GATGCGAATTCAGTACG$"         [4]    := "ACGGATGCGAATTCAGTACG$"
            "TACGGATGCGAATTCAGTACG$"     [0]    := "ACGTACGGATGCGAATTCAGTACG$"

       "T" →
            "GCGAATTCAGTACG$"            [8]    := "ATGCGAATTCAGTACG$"
            "TCAGTACG$"                  [14]   := "ATTCAGTACG$"

       "AATTCAGTACG$"                    [13]   := "AATTCAGTACG$"
       "GTACG$"                          [18]   := "AGTACG$"

  "C" →
```

```
     "A" →
          "GTACG$"                    [17]   := "CAGTACG$"

     "G" →
           "$"                         [22]   := "CG$"
           "AATTCAGTACG$"              [11]   := "CGAATTCAGTACG$"
           "GATGCGAATTCAGTACG$"        [5]    := "CGGATGCGAATTCAGTACG$"
           "TACGGATGCGAATTCAGTACG$"    [1]    := "CGTACGGATGCGAATTCAGTACG$"

"G" →
     "A" →
          "ATTCAGTACG$"              [12]   := "GAATTCAGTACG$"
          "TGCGAATTCAGTACG$"         [7]    := "GATGCGAATTCAGTACG$"

     "CGAATTCAGTACG$"                 [10]   := "GCGAATTCAGTACG$"
     "GATGCGAATTCAGTACG$"             [6]    := "GGATGCGAATTCAGTACG$"
     "GT" →
          "ACG$"                      [19]   := "GTACG$"
          "ACGGATGCGAATTCAGTACG$"   [2]    := "GTACGGATGCGAATTCAGTACG$"
     "$"                              [23]   := "G$"

"T" →
     "ACG" →
          "$"                         [20]   := "TACG$"
          "G GATGCGAATTCAGTACG$"    [3]    := "TACGGATGCGAATTCAGTACG$"

     "GCGAATTCAGTACG$"                [9]    := "TGCGAATTCAGTACG$"
     "CAGTACG$"                       [16]   := "TCAGTACG$"
     "TCAGTACG$"                      [15]   := "TTCAGTACG$"

"$"                                   [24]   := "$"
```

Notes about this layout:

- The top-level branches from the root are one per distinct first character: `A`, `C`, `G`, `T`, and `$`.
- Each compressed edge label is the maximal substring from T that is common along that path before a branching point (or the remainder of a suffix).
- Leaves are annotated with the suffix start index (0-based). The right-hand comment after each leaf (written as `":= '...'"`) shows the full suffix string for clarity.
- Where a small internal node is necessary (several suffixes share a longer prefix), that interior node is shown, and its outgoing edges list the exact remainders.

8.3.1 Search Process (Threading the Pattern)

We now search for $P =$ `GAATTC` in the Suffix Tree.

Step 1. Start at Root: `pattern = "GAATTC"`

- Look for an edge beginning with `'G'`.
- Found edge `"G"` leading to an internal node (call it `Node_G`).
- Pattern still remains (`"AATTC"`).

- **Action:** Follow the edge to `Node_G`.

Step 2. At Node_G: `remaining_pattern = "AATTC"`

- Outgoing edges from `Node_G` begin with `'A'`, `'C'`, `'G'`, etc.
- Choose edge `"A"` leading to `Node_GA`.
- Remaining pattern: `"ATTC"`.
- **Action:** Follow edge `"A"`.

Step 3. At Node_GA: `remaining_pattern = "ATTC"`

- Among outgoing edges from `Node_GA`, we find one labeled `"ATTCAGTACG$"`.
- Compare character by character: `"ATTC"` matches the prefix of `"ATTCAGTACG$"`.
- Pattern completely consumed.
- **Action:** Return `true`.

8.3.2 Mismatch Example (Pattern `"GATTAG"`)

Step 1. At Root: Find edge `"G"` $\Rightarrow$ go to `Node_G`, pattern becomes `"ATTAG"`.

Step 2. At Node_G: Find edge `"A"` $\Rightarrow$ go to `Node_GA`, pattern becomes `"TTAG"`.

Step 3. At Node_GA: Attempt to match edge `"ATTCAGTACG$"` with pattern `"TTAG"`. Mismatch occurs at second character (`'T'` $\neq$ `'A'`). **Action:** Return `false`.

8.3.3 Key Observations

- Each edge traversal compares one or more characters, with total comparisons equal to $|P| = 6$.
- Search complexity is $O(m)$, independent of the text length n.
- The pattern `GAATTC` is successfully found via path: `(Root)` $\rightarrow$ `G` $\rightarrow$ `A` $\rightarrow$ `ATTCAGTACG$`.
- Unlike binary search in a Suffix Array, no $\log n$ factor appears.
- The trade-off: the Suffix Tree's large constant space cost and construction complexity versus its $O(m)$ query time.