FINAL PROJECT

Comparative Analysis and Implementation of Advanced String-Matching Algorithms for Large-Scale DNA Sequence Processing: General-Purpose to DNA-Optimized Approaches.

Course: CS 5800, Algorithms.

Instructor:

Marko Puljic

Team Members:

Group – 12,
Joel Markapudi – NUID: 002471866,
Ryan Denny – NUID: 002444870,
Jennisha Christina Martin – NUID: 002415325,
Tanay Grover – NUID: 002470621.

INDEX:

Topic Introduction:

DNA pattern matching is a crucial task in genomics, involving the search for specific genetic sequences within vast DNA datasets. As genomic data grows exponentially, traditional string-matching algorithms struggle to keep up. This project tackles this challenge by exploring advanced approaches: EPMSPP, Ukkonen's Suffix Tree, KMP, Rabin-Karp, Boyer-Moore, and Aho-Corasick. We're not just implementing these; we're putting them through their paces in the DNA context.

Our key question is: How do these algorithms perform in terms of efficiency and scalability when processing large-scale DNA sequences? We're implementing and testing these methods to understand their behavior under various conditions of sequence length and pattern complexity. Our rationale is simple: by improving DNA pattern matching, we can accelerate genomic research, potentially leading to breakthroughs in personalized medicine and genetic disease studies. We're doing this because faster, more efficient DNA analysis tools can have a real impact on human health and our understanding of life itself. It's not just about algorithms; it's about paving the way for discoveries.

Formal Problem Definition:

Given a DNA sequence $S = s_1s_2...s_n$ of length n over the alphabet $\Sigma = \{A, C, G, T\}$, and a pattern $P = p_1p_2...p_m$ of length m, where $6 \leq m \leq k < n$, we aim to determine the set of all occurrences of P in S. We are dealing with pattern sizes in general from $6 \leq m \leq 15$ for practice, and in general - $6 \leq m \leq k$, but not limited.

Formally, we seek to find the set O defined as:

$O = \{i \mid 1 \leq i \leq n - m + 1 \text{ and } S[i, i+m-1] = P\}$
where S[i, j] denotes the substring of S from position i to position j, inclusive. n - m + 1 is the last position where P could possibly start and still fit within S.

The statement means, O is the **set of all positions** i in S where, if you start at i and take m characters, you get an exact match of the pattern P. The cardinality of set O, |O|, represents the number of exact matches of P in S.

Constraints:

1. $\forall s_i \in S, s_i \in \Sigma$
   - (every character s_i in the sequence S must be an element of the alphabet Σ.)
2. $\forall p_j \in P, p_j \in \Sigma$
   - (every character p_j in the pattern P must also be an element of Σ.)
3. $6 \leq m \leq k < n$

The objective is to implement an efficient algorithm A that, given S and P as input, outputs the set O (or |O|) while minimizing the time complexity f(n, m) and space complexity g(n, m).

Data Source and Generation:

For this project, we utilized a custom DNA sequence generator to create synthetic genomic data. This approach allows for the creation of large-scale DNA sequences with controllable characteristics, enabling thorough testing of our algorithms under various conditions. The DNA generator produces sequences using the four nucleotide bases: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). It generates random sequences of specified lengths, simulating the complexity and variability found in real genomic data. This method ensures a consistent and reproducible testing environment while allowing for scalability in sequence lengths. Here's a brief pseudocode of our DNA generator:

```
Function GenerateDNASequence(length):
    bases = ['A', 'C', 'G', 'T']
    sequence = ""
    for i = 1 to length:
        sequence += RandomChoice(bases)
    return sequence
```

CLRS Textbook / Course Modules Reference:

The algorithms implemented in this project draw inspiration from the fundamental concepts presented in *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein (CLRS). While the textbook doesn't specifically cover DNA sequence matching algorithms, it provides crucial insights into string matching techniques and data structures that form the foundation of our approach. The Knuth-Morris-Pratt (KMP) algorithm discussed in CLRS influenced our thinking on efficient pattern matching, while the concepts of suffix trees and their construction, although not directly used in EPMSPP, informed our understanding of advanced string processing techniques. The analysis methods presented in CLRS, particularly for time and space complexity, guided our performance evaluations and optimizations for DNA-specific applications.

Scope of the Project Solutions:

This project encompasses a comprehensive analysis and implementation of various string-matching algorithms, with a specific focus on their application to DNA sequence analysis. The scope includes both general-purpose algorithms and those specifically optimized for genomic data. We have implemented and analyzed the Exact Multiple Pattern Matching Algorithm using DNA Sequence and Pattern Pair (EPMSPP), Ukkonen's Suffix Tree Algorithm, Rabin-Karp Algorithm, Z-Algorithm, Knuth-Morris-Pratt (KMP) Algorithm, Boyer-Moore Algorithm, and Aho-Corasick Algorithm. Our solutions cover a wide range of approaches, from hash-based methods to suffix trees and finite automata, providing a broad perspective on string matching techniques applicable to bioinformatics and beyond.

Clearly Defined Questions:

1. How do DNA-specific algorithms like EPMSPP compare in efficiency to general-purpose string-matching algorithms when applied to genomic sequences?
2. What are the trade-offs between preprocessing time, query time, and memory usage among the different algorithms, particularly in the context of large DNA sequences?
3. How do the algorithms perform under varying conditions such as pattern length, text length, and alphabet size, specifically in the context of DNA (4-letter alphabet)?
4. Which algorithms are best suited for single pattern matching versus multiple pattern matching in genomic data?
5. How do the theoretical time complexities of these algorithms translate to practical performance in real-world DNA sequence analysis tasks?
6. What are the scalability characteristics of each algorithm when dealing with very large genomic datasets?

Personal Context:

- Joel focused on EPMSPP and Ukkonen's Algorithm, driven by a fascination with DNA-specific optimizations and the elegant theoretical foundations of suffix trees.
- Ryan explored the Rabin-Karp algorithm, intrigued by its hash-based approach and its potential applications in plagiarism detection.
- Tanay delved into the Z-algorithm, attracted by its unique prefix-based analysis and its applications in compressed string matching.
- Jennisha tackled the Aho-Corasick, KMP, and Boyer-Moore algorithms, motivated by their widespread use in text processing and their varying approaches to efficient pattern matching.

Introduction to Algorithms:

EPMSPP Algorithm: The Exact Multiple Pattern Matching Algorithm using DNA Sequence and Pattern Pair (EPMSPP) is an approach to DNA pattern matching. It leverages the unique properties of DNA sequences to optimize search efficiency. The algorithm's core idea is to create index tables for both the sequence and pattern, focusing on pairs of nucleotides rather than individual bases. This pairing approach reduces the search space significantly, as there are 16 possible pairs (4^2) compared to 4 individual bases. The algorithm then identifies the least frequent pair in the pattern to minimize unnecessary comparisons, enhancing performance, for large DNA sequences.

Ukkonen's Algorithm: Ukkonen's Algorithm is a linear-time method for constructing suffix trees, which are powerful data structures for pattern matching in strings. In the context of DNA sequence analysis, this algorithm provides an efficient way to build a comprehensive index of all substrings within a DNA sequence. The key innovation of Ukkonen's approach is its online nature – it processes the input string from left to right in a single pass, updating the suffix tree incrementally. This makes it particularly suitable for real-time applications or scenarios where the full sequence is not immediately available. Once constructed, the suffix tree allows for rapid pattern matching, supporting various genomic analyses like finding repeats, locating specific sequences, or identifying common substrings.

Z-Algorithm: The Z-Algorithm is an efficient string-matching algorithm designed to find all occurrences of a pattern within a text. The key concept behind the Z-Algorithm is the construction of a Z-array, where each element in the array represents the length of the longest substring starting from that position that matches the prefix of the text. By leveraging this Z-array, the algorithm can quickly identify potential matches without redundant comparisons, offering a linear-time solution for pattern matching problems. In the context of DNA sequence analysis, the Z-Algorithm is particularly useful due to its ability to handle large datasets efficiently. It is well-suited for tasks such as identifying motifs, searching for specific nucleotide patterns, and detecting repeating sequences within genomic data. The simplicity of the Z-Algorithm, combined with its powerful pattern matching capabilities, makes it a valuable tool for bioinformatics and other applications that require precise and fast string matching.

Rabin-Karp Algorithm: The Rabin-Karp algorithm is a classical string-matching algorithm particularly suited for multiple pattern matching. It leverages hashing to find any one of a set of pattern strings in a text. By converting a string into a numerical hash value, it can efficiently locate patterns within a large DNA sequence by comparing these hash values rather than the strings directly. This is particularly beneficial in scenarios like DNA sequence analysis, where the patterns are fixed, and the text (DNA sequence) is massive. However, the algorithm has limitations, including the potential for hash collisions, which can lead to false positives. The choice of a good hash function is critical to minimizing these collisions. Additionally, the algorithm's performance can degrade if many hash values need to be recalculated, particularly in scenarios with frequent character changes.

Knuth-Morris-Pratt (KMP): The Knuth-Morris-Pratt (KMP) algorithm is an efficient string-matching method particularly useful for DNA sequence analysis. Its key feature is a preprocessing step that constructs a failure function, also known as the Longest Proper Prefix which is also Suffix (LPS) array. This preprocessing allows the algorithm to skip unnecessary comparisons during the matching phase, resulting in a linear time complexity of $O(n + m)$ for a text of length n and a pattern of length m. KMP is especially effective for sequences with repetitive substrings, a common characteristic in genetic data. The algorithm's ability to handle large datasets through stream processing makes it valuable for analysing extensive genomic sequences. While primarily used for exact pattern matching, the LPS array generated during pre-processing can also provide insights into the internal structure of DNA patterns.

Boyer-Moore: The Boyer-Moore algorithm is a string searching method that improves efficiency by skipping characters based on information gathered from mismatches. It employs two main heuristics: the bad character rule and the good suffix rule. The bad character rule shifts the pattern to align the mismatched character in the text with its rightmost occurrence in the pattern. The good suffix rule shifts the pattern to align a matching suffix with its next occurrence. These heuristics are implemented using pre-processed tables. The algorithm compares characters from right to left and can skip large portions of the text, leading to sublinear time complexity in practice, though the worst-case remains O(nm). This efficiency makes Boyer-Moore particularly suitable for searching large genomic datasets where the pattern is relatively rare within the sequence. Its effectiveness increases with longer pattern lengths, making it well-suited for identifying specific genes or long DNA motifs.

Aho-Corasick: The Aho-Corasick algorithm is designed for efficient multiple pattern matching. It constructs a finite automaton from the set of search patterns, creating a trie-like structure augmented with failure links. This allows for simultaneous matching of multiple patterns in a single pass through the text, with a time complexity of O(n + m + k), where n is the length of the text, m is the total length of all patterns, and k is the number of pattern occurrences. The algorithm is particularly useful in genomics for identifying multiple DNA motifs or gene sequences simultaneously. Its linear-time complexity ensures scalability with large genomic datasets and numerous search patterns. The Aho-Corasick algorithm's ability to match multiple patterns efficiently makes it valuable in comparative genomics and for identifying complex genetic patterns that may involve multiple sequence motifs.

Algorithm Approach / Pseudocodes And Analysis:

EPMSPP Algorithm:

```
1. Create index tables for sequence and pattern, storing positions of each nucleotide pair.
2. Find the least frequent pair in the pattern that occurs in the sequence.
3. For each occurrence of the least frequent pair in the sequence:
   a. Calculate potential match position
   b. Validate position (non-negative and enough room for full pattern)
4. Compare all pairs in the pattern at the potential match position:
   a. Start with most frequent pairs for early mismatch detection
   b. Use pre-computed index tables for quick lookups
5. If all pairs match, add position to matches list.
6. Optimize comparison order based on pattern characteristics.
7. Return list of match positions.
```

<u>Time And Space Complexity Analysis (EPMSPP Algorithm):</u>

1. Index Table Creation (createIndexTable method):
    - Time: O(n) for sequence, O(m) for pattern
    - Where n is the length of the sequence, m is the length of the pattern
    - Iterates once through the input string, constant-time operations per character
2. Pair Index Computation (computePairIndex method):
    - Time: O(1), Constant time bitwise operations
3. Finding Least Frequent Pair (findLeastFrequentPair method):
    - Time: O(ALPHABET_SIZE + m + n), ALPHABET_SIZE is constant (16 for DNA pairs)
    - Iterates through all pairs, counting occurrences in pattern and sequence
4. Main Matching Logic:
    - Worst-case Time: O(n * m) , Best-case Time: O(n/m)
        - Iterates through occurrences of least frequent pair: O(n) worst case
        - For each occurrence, compares pairs in the pattern: O(m)
        - Best case occurs when the least frequent pair is rare in the sequence
5. Overall Time Complexity:
    - Preprocessing: O(n + m) for creating index tables. Matching: O(n * m) worst case, **O(n/m) best case.**
    - Total: O(n + m + n * m) = O(n * m) worst case

The average case is often better than O(n * m), especially for DNA sequences where the distribution of bases is relatively uniform.

6. Index Tables:
    - Space: O(n + m), Two 2D arrays of size ALPHABET_SIZE * max(n, m)
    - ALPHABET_SIZE is constant (16)
7. Output Storage:
    - Space: O(n) worst case, for storing all match positions (if pattern occurs frequently)
8. Overall Space Complexity: **O(n + m)**

<u>Final Complexity Analysis:</u>

- For time complexity: Average-case: Often better than O(n + m) for typical DNA sequences and, Best-case: O(n/m).
- For Space Complexity: O(n + m)

<u>Key Points:</u> This efficiency comes from the pair indexing system and the use of the least frequent pair strategy. The algorithm performs particularly well when the least frequent pair in the pattern is rare in the sequence, allowing it to skip large portions of the sequence. For DNA sequences, where the alphabet is small and relatively uniformly distributed, the algorithm often performs closer to its best-case time complexity. The space complexity is linear with respect to the input sizes. In practice, for typical DNA sequences, this algorithm often outperforms traditional string-matching algorithms, especially for longer sequences and shorter patterns.

Ukkonen's Algorithm:

1. Initialize suffix tree with root node and set up active point.
2. For each character in the input string:
   a. Increment global end for all leaf nodes
   b. Add new suffixes while updating active point
3. While adding suffixes:
   a. Create new leaf if suffix doesn't exist
   b. Split edge and create leaf if partial match
   c. Update suffix links for newly created internal nodes
4. Update active point:
   a. Follow suffix link if not at root
   b. Slide down if at root with remaining active length
5. Optimize for space using integer ranges for edges instead of substrings.
6. To find a pattern, traverse from root following matching edges:
   a. If full match, collect all leaf positions in subtree
   b. Handle partial matches ending mid-edge
7. Consider DNA-specific optimizations like 2-bit character encoding.

Time and Space Complexity Analysis (Ukkonen's Algorithm):

1. Suffix Tree Construction: Overall Time: O(n) for a string of length n
   - Each of the n phases takes amortized O(1) time
2. Key Operations: a. Extending the tree (extendSuffixTree method):
   - Amortized Time: O(1) per character. Total Time: O(n) for n characters

b. Walk Down Procedure (walkDown method): Time: O(1) amortized
c. Edge Length Calculation (edgeLength method): Time: O(1)

3. Pattern Matching (findPattern method):
   - Time: O(m + k), where m is pattern length and k is number of occurrences
   - Traversal: O(m), Collecting matches: O(k)

Space Complexity

1. Suffix Tree Structure:
   - Space: O(n) for a string of length n
   - Worst case: O(n^2) if not using path compression (rarely occurs in practice)
2. Key Components: a. Text Storage: Space: O(n) for byte array representation

b. Nodes: Space: O(n) total nodes. Each node: O(1) space (fixed-size array for children, constant number of fields)
c. Suffix Links: Space: O(n), one per internal node.

3. Additional Data Structures: activeNode, activeEdge, activeLength: O(1), remainingSuffixCount, leafEnd, position: O(1).
   - nodes List: O(n) for storing all nodes

Final Complexity:

- Time Complexity: O(n) for construction, O(m + k) for pattern matching
- Space Complexity: O(n) in practice, O(n^2) worst case (rare)

Ukkonen's algorithm achieves linear time complexity through clever use of suffix links and the "active point" concept, making it efficient for large-scale DNA sequence analysis. The space efficiency makes it practical for handling substantial genomic datasets.

Important Concepts For Suffix Tree:

1. Suffix Link:
   - A suffix link is a pointer from an internal node representing a string αA (where A is a single character and α is a string) to another internal node representing string α.
   - Suffix links are crucial for the efficiency of Ukkonen's algorithm but are not typically used in the final representation.
2. Edge-Label Compression:
   - Edge labels are typically represented by start and end indices in the original string, not by storing the actual substring. This saves space for long, repetitive sequences.
3. $ Terminator:
   - Often, a special terminator character (usually $) is appended to the string to ensure all suffixes end at leaf nodes.This guarantees that no suffix is a prefix of another suffix, simplifying certain operations.
4. Implicit vs. Explicit Nodes:
   - Explicit nodes are actually present in the tree structure.
   - Implicit nodes are conceptual nodes that would exist if the tree were fully expanded but are not actually created to save space.
5. Generalized Suffix Tree:
   - A suffix tree can be built for multiple strings by using distinct terminators for each string. This allows for operations across multiple strings.
6. Leaf Node Properties:
   - Each leaf corresponds to a suffix of the string.
   - The depth of a leaf node from the root corresponds to the length of the suffix it represents.
7. Lowest Common Ancestor (LCA): The LCA of two leaves represents the longest common prefix of the suffixes they represent. This property is useful for various string processing tasks.
8. Path Label: The concatenation of all edge labels from the root to a node is called the path label of that node. For any internal node, its path label is a prefix of all leaf nodes in its subtree.
9. Suffix Tree Depth: The depth of a suffix tree is O(n) where n is the length of the string. This is because the longest possible suffix (the entire string) has length n.
10. Right-to-Left Construction: Some implementations build the tree from right to left, which can simplify certain aspects of the construction.
11. Suffix Numbers: Often, leaves are annotated with the index where their corresponding suffix begins in the original string. This allows quick location of suffixes in the original string.
12. Implicit Suffix Tree:
    - In Ukkonen's algorithm, **intermediate trees that don't explicitly represent all suffixes** but can derive them are called implicit suffix trees. The final tree after processing all characters is an explicit suffix tree.

Understanding these concepts will give you a more comprehensive view of suffix trees and how they represent string data. They're particularly important when implementing or working with suffix trees in practice, as they influence both the structure of the tree and the algorithms that operate on it.

Z-Algorithm:

1. **Initialize Z-array** with the length of the input string, setting the first element to the string length.
2. **Set left and right boundaries** for the current Z-box, starting both at index 0.
3. **Iterate over each character** in the string from the second position onward: a. If the current index is outside the Z-box, perform a direct comparison from the current position, updating the Z-box boundaries accordingly. b. If the current index is within the Z-box, use previously computed Z-values to determine a potential match length without direct comparison.
4. **Adjust Z-box boundaries** when a match is extended beyond the current Z-box, expanding the box as needed.
5. **Store match lengths** in the Z-array, corresponding to the longest prefix match starting at each position.
6. **Use the Z-array** to find pattern occurrences by comparing the pattern with sections of the text: a. A full pattern match is indicated when a Z-value equals the pattern length.
7. **Optimize the search** by skipping unnecessary comparisons and leveraging the Z-box for efficient matching.
8. **Return the list of positions** where the pattern occurs in the text, based on the Z-array values.

Rabin-Karp Algorithm:

1. **Hash Function Initialization:**
   - Compute the hash value of the pattern and the first substring of the text (of length equal to the pattern).
   - Use a rolling hash function to update the hash value of the subsequent substrings.
2. **Pattern Matching:**
   - For each substring in the text, compare its hash value to the hash value of the pattern.
   - If the hash values match, perform a direct string comparison to verify the match.
3. **Sliding Window Hash Update:**
   - Update the hash value by subtracting the leading character and adding the trailing character.
4. **Handle Collisions:**
   - If the hash values match but the strings don’t, it’s a collision. This is resolved by confirming the actual strings.
5. **Return Matches:**
   - If a match is found, store the position of the match.

KMP (Knuth-Morris-Pratt) Algorithm:

1. Preprocess the Pattern to Create the LPS Array:
   a. Initialize the LPS (Longest Prefix Suffix) array with zeros.

b. Use two pointers (len and i) to compare characters in the pattern.
c. Build the LPS array by comparing matching prefixes and suffixes:
   I. If characters match, increment both pointers and set the LPS value.
   II. If characters do not match and len is not zero, update len to LPS[len-1].
   III. If characters do not match and len is zero, set LPS[i] to zero and increment i.

2. Search the Text Using the Pre-processed Information:
   a. Initialize text pointer i and pattern pointer j to zero.
   b. Compare characters of the pattern and text:
      I. If characters match, increment both i and j.
      II. If j equals the length of the pattern (a match is found), record the position, reset j using LPS[j-1] and continue.
      III. If characters do not match and j is not zero, reset j using LPS[j-1].
      IV. If characters do not match and j is zero, increment i.
   c. Repeat until the end of the text is reached.

3. Return the List of Match Positions:
   After the search is complete, return the list of recorded match positions.

<u>Boyer-Moore Algorithm:</u>

1. Preprocess the Pattern:
   a. Create the bad character heuristic table.
   b. Create the good suffix heuristic table.
2. Search the Text:
   a. Start comparison from the rightmost character of the pattern.
   b. If characters match, move left in the pattern.
   c. If a mismatch occurs:
      i. Use the bad character heuristic to calculate the shift.
      ii. Use the good suffix heuristic to calculate the shift.
      iii. Take the maximum of the two shifts and move the pattern accordingly.
3. Match Handling:
   a. If a complete match is found, record the position.
   b. Continue the search from the new position.
4. Continue the Process:
   Repeat steps 2-3 until the pattern reaches the end of the text.
5. Return Results:
   Return the list of recorded match positions.

<u>Aho-Corasicks Algorithm:</u>

1. Construct the Trie (Prefix Tree) from the Set of Patterns:
   a. Create the root node of the trie.
   b. For each pattern in the set:
      I. Add its characters sequentially to the trie, creating new nodes as needed.
      II. Mark the end of the pattern in the trie (this node will hold the output of the pattern).
2. Build Failure Links:
   a. Use a breadth-first search (BFS) to traverse the trie starting from the children of the root.
   b. For each node:

        I. Set its failure link to the node that represents the longest proper suffix of the pattern represented by the current node.
        II. If no such suffix exists, set the failure link to the root.

3. Construct Output Links:
    a. Traverse the trie again.
    b. For each node, if the node reached by the failure link has an output (it represents a suffix that is also a pattern), add an output link from the current node to that node.
4. Search the Text Using the Constructed Automaton:
    a. Start from the root of the trie.
    b. For each character in the text:
        i. If a transition (child node) exists for the character, follow it.
        ii. If no transition exists, follow the failure links until a valid transition is found or return to the root.
        iii. If the current node has an output, report matches for all patterns linked to that node.
5. Return the list of all match positions for all patterns.

Algorithm Result Collection:

EPMSPP Algorithm:

Metric	Test 1	Test 2	Test 3
Sequence Length	100,000,000	100,000,000	100,000,000
Sequence Generation Time (s)	0.9549613	0.9543269	0.9872001
Pattern 1	GCTTGGGTAC	GAACGTGAGG	TCCGCATACA
Pattern 1 Matches	92	91	91
Pattern 1 Matching Time (s)	3.343375	3.296597	3.368703
Pattern 1 Matching Speed (Mbp/s)	29.91	30.33	29.69
Pattern 2	CCAGTGGAATGA	AGGAGCAAGTCT	AGTCATAACTAC
Pattern 2 Matches	6	11	6
Pattern 2 Matching Time (s)	1.828532	1.846105	1.805958
Pattern 2 Matching Speed (Mbp/s)	54.69	54.17	55.37
Pattern 3	CACAACACAGCTTC	CTACTAAACTTTGC	CCCTTGATCGCATA
Pattern 3 Matches	0	1	0
Pattern 3 Matching Time (s)	2.022476	1.902855	2.032230
Pattern 3 Matching Speed (Mbp/s)	49.44	52.55	49.21
Total Pattern Matching Time (s)	7.194382	7.045556	7.206891
Average Matching Time per Pattern (s)	2.398127	2.348519	2.402297
Average Matching Speed (Mbp/s)	41.70	42.58	41.63
Total Execution Time (s)	7.207384	7.058694	7.219420

Ukkonen's Algorithm:

Metric	Test 1	Test 2	Test 3
Sequence Length	1,000,000	1,000,000	1,000,000
Sequence Generation Time (s)	0.017233	0.015949	0.017894
Suffix Tree Construction Time (s)	178.328941	174.986773	307.267290
Pattern 1	GAACATAGTG	CCAGAAGACA	CGTTACCTCA
Pattern 1 Matches	3	2	1
Pattern 1 Matching Time (s)	0.000295	0.000307	0.001238
Pattern 2	TACGGAAAGCACCAG	ATGTTCTGTTTTCTT	GCGCCATTTTGGAGT
Pattern 2 Matches	0	0	0
Pattern 2 Matching Time (s)	0.000028	0.000025	0.000114
Total Pattern Matching Time (s)	0.000324	0.000332	0.001351
Average Matching Time per Pattern (s)	0.000162	0.000166	0.000676
Total Execution Time (s)	178.363514	175.021269	307.319093

Rabin Karp Algorithm:

Metric	Test 2	Test 3	Test 4	Test 5	Test 6
Sequence Length	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
Pattern 1	ACG	ACG	ACG	ACG	ACG
Pattern 1 Matches	1513	15,702	155,777	1,563,112	15,626,598
Pattern 1 Matching Time (s)	0.0053112	0.0071846	0.055929899	0.5285115	5.3250051
Matching Speed (Mbp/s)	18.83 Mbp/s	139.19 Mbp/s	178.80 Mbp/s	189.21 Mbp/s	187.79 Mbp/s
Pattern 2	ACGTACGT	ACGTACGT	ACGTACGT	ACGTACGT	ACGTACGT
Pattern 2 Matches	1	20	178	1,536	15,117
Pattern 2 Matching Time (s)	0.002886501	0.015589	0.0380018	0.3876108	3.771242199
Matching Speed (Mbp/s)	34.64 Mbp/s	64.15 Mbp/s	263.15 Mbp/s	257.99 Mbp/s	265.16 Mbp/s
Pattern 3	ACGTACGT ACGTACGT	ACGTACGT ACGTACGT	ACGTACGT ACGTACGT	ACGTACGT ACGTACGT	ACGTACGT ACGTACGT
Pattern 3 Matches	0	0	0	0	1

Pattern 3 Matching Time (s)	3.65399E-4	0.0152598	0.0410414	0.393172299	3.830801
Matching Speed (Mbp/s)	273.67 Mbp/s	65.53 Mbp/s	243.66 Mbp/s	254.34 Mbp/s	261.04 Mbp/s

Knuth-Morris-Pratt Algorithm (KMP):

Metric	Test 1	Test 2	Test 3
Sequence Length	100,000,000	100,000,000	100,000,000
Sequence Generation Time (s)	0.000141 s	0.000000 s	0.000001 s
Pattern 1	TGAACAACGC	ATAATGGCAT	TCCTGAGTAT
Pattern 1 Matches	93	88	92
Pattern 1 Matching Time (s)	0.834426 s	0.797146 s	0.784607 s
Pattern 1 Matching Speed (Mbp/s)	119.84 Mbp/s	125.45 Mbp/s	127.45 Mbp/s
Pattern 2	CGCTGAACTA	TCCTTTAGCA	GAGCCCACCT
Pattern 2 Matches	101	79	105
Pattern 2 Matching Time (s)	0.756734 s	0.790928 s	0.753761 s
Pattern 2 Matching Speed (Mbp/s)	132.15 Mbp/s	126.43 Mbp/s	132.67 Mbp/s
Pattern 3	TTTTGCTATG	GTTAGCCACT	CCTGTCGCTT
Pattern 3 Matches	77	103	92
Pattern 3 Matching Time (s)	0.717391 s	0.779271 s	0.732130 s
Pattern 3 Matching Speed (Mbp/s)	139.39 Mbp/s	128.33 Mbp/s	136.59 Mbp/s
Total Pattern Matching Time (s)	2.308551 s	2.367345 s	2.270498 s
Average Matching Time per Pattern (s)	0.769517 s	0.789115 s	0.756833 s
Average Matching Speed (Mbp/s)	129.95 Mbp/s	126.72 Mbp/s	132.13 Mbp/s
Total Execution Time (s)	2.309386 s	2.367440 s	2.270584 s

Boyer-Moore Algorithm:

Metric	Test 1	Test 2	Test 3
Sequence Length	100,000,000	100,000,000	100,000,000
Sequence Generation Time (s)	0.000141 s	0.000000 s	0.000001 s

Pattern 1	TGAACAACGC	ATAATGGCAT	TCCTGAGTAT
Pattern 1 Matches	87	31	102
Pattern 1 Matching Time (s)	0.594712 s	0.723456 s	0.627981 s
Pattern 1 Matching Speed (Mbp/s)	168.15 Mbp/s	138.23 Mbp/s	159.24 Mbp/s
Pattern 2	CGCTGAACTA	TCCTTTAGCA	GAGCCCACCT
Pattern 2 Matches	114	97	100
Pattern 2 Matching Time (s)	0.690866 s	0.709856 s	0.606212 s
Pattern 2 Matching Speed (Mbp/s)	144.75 Mbp/s	140.87 Mbp/s	164.96 Mbp/s
Pattern 3	TTTTGCTATG	GTTAGCCACT	CCTGTCGCTT
Pattern 3 Matches	19	92	81
Pattern 3 Matching Time (s)	0.662457 s	0.651429 s	0.566592 s
Pattern 3 Matching Speed (Mbp/s)	150.95 Mbp/s	153.51 Mbp/s	176.49 Mbp/s
Total Pattern Matching Time (s)	1.948034 s	2.084741 s	1.800785 s
Average Matching Time per Pattern (s)	0.649345 s	0.694914 s	0.600262 s
Average Matching Speed (Mbp/s)	154.00 Mbp/s	143.90 Mbp/s	166.59 Mbp/s
Total Execution Time (s)	1.948417 s	2.084827 s	1.800901 s

Aho-Corasick Algorithm:

Metric	**Test 1**	**Test 2**	**Test 3**
Sequence Length	100,000,000	100,000,000	100,000,000
Sequence Generation Time (s)	0.000141 s	0.000001 s	0.000001 s
Patterns	[TCGGCTCGAG, GGCAATATGT, TGGAGCTCAT]	[AAAAAGTTGA, TTCTATACCA, TTTAGTTCAG]	[TTGCACAGCA, GCCAAAGGAA, GGAAAATGCA]
Total Matches	318	290	304
Total Pattern Matching Time	1.577368 s	1.390056 s	1.507911 s
Matching Speed	63.40 Mbp/s	71.94 Mbp/s	66.32 Mbp/s
Trie size	30	29	30
Total Execution Time	1.584010 s	1.390795 s	1.508491 s

Z-Algorithm:

Metric	Test 1	Test 2	Test 3
Sequence Length	100,000	1,000,000	10,000,000
Sequence Generation Time (ms)	47	38	213
Pattern 1	ACGT	ACGT	ACGT
Pattern 1 Matches	385	3,917	38,538
Pattern 1 Matching Time (ms)	95	11	558
Pattern 1 Matching Speed (MB/s)	1.00	86.70	17.09
Pattern 2	GGTA	GGTAC	GGTAC
Pattern 2 Matches	130	961	9,579
Pattern 2 Matching Time (ms)	3	49	121
Pattern 2 Matching Speed (MB/s)	31.79	19.46	78.82
Pattern 3	TATA	TATA	TATA
Pattern 3 Matches	395	3,845	39,377
Pattern 3 Matching Time (ms)	8	105	172
Pattern 3 Matching Speed (MB/s)	11.92	9.08	55.45

ALGORITHM ADVANTAGES:

Specific advantages of EPMSPP:

1. **DNA-Specific Optimization:** EPMSPP is designed specifically for DNA sequences, taking advantage of the limited alphabet (A, C, G, T) to create efficient indexing.
2. **Pair-Based Approach:** By using pairs of characters instead of single characters, EPMSPP reduces the search space significantly. This is particularly effective for DNA sequences where the combination of pairs is limited to 16 possibilities.
3. **Multiple Pattern Matching:** EPMSPP can handle multiple patterns efficiently, which is a significant advantage over algorithms like KMP or Boyer-Moore that are designed for single pattern matching.
4. **Reduced Comparisons:** The use of the least frequent pair in the pattern for initial matching significantly reduces the number of unnecessary comparisons, especially in long sequences.
5. **Linear Time Complexity:** In both average and worst cases, EPMSPP maintains linear time complexity, which is competitive with KMP and better than Boyer-Moore's worst case.

Specific advantages of Ukkonen's:

1. **Linear Time Complexity:** Ukkonen's algorithm constructs the suffix tree in O(n) time, where n is the length of the input string, making it highly efficient for large DNA sequences.
2. **Online Construction:** The algorithm builds the suffix tree incrementally as it reads the input, allowing for real-time processing of DNA sequences as they are generated.

3. **Space Efficiency:** By using clever techniques like suffix links and a single-phase approach, Ukkonen's algorithm minimizes the space overhead typically associated with suffix trees.
4. **Versatile Pattern Matching:** Once constructed, the suffix tree allows for rapid matching of multiple patterns of varying lengths, making it ideal for diverse genomic analyses.
5. **Substring Identification:** The suffix tree structure enables quick identification of all substrings in the sequence, facilitating analyses like finding repeats or common subsequences.
6. **Adaptability to DNA-specific Optimizations:** The algorithm can be easily adapted to use 2-bit encoding for DNA bases, further improving space efficiency and processing speed.
7. **Support for Advanced Analyses:** Beyond simple pattern matching, the suffix tree supports more complex operations like finding the longest common substring or all palindromes in the sequence.

<u>Specific advantages of Z-algorithm:</u>

1. **Linear Time Complexity:** The Z-Algorithm processes the input string in O(n) time, where n is the length of the string, making it highly efficient for pattern matching tasks, especially in large DNA sequences.
2. **Simplicity and Efficiency:** The algorithm's straightforward approach, based on constructing the Z-array, allows for quick and efficient pattern matching without the need for complex data structures, making it easy to implement and understand.
3. **Optimal Substring Search:** By leveraging the Z-array, the algorithm identifies all occurrences of a pattern within a text without redundant comparisons, optimizing the search process, particularly in repetitive or structured DNA sequences.
4. **Efficient Prefix Matching:** The Z-Algorithm excels at finding the longest prefix match starting from any position in the string, which is particularly useful in bioinformatics applications where prefix patterns are common.
5. **Versatility for Multiple Patterns:** The algorithm can be adapted to handle multiple patterns searches efficiently, making it suitable for genomic analyses where multiple motifs or sequences need to be identified simultaneously.
6. **Low Space Overhead:** Unlike more complex algorithms that require additional data structures, the Z-Algorithm only requires an additional array of the same length as the input string, resulting in minimal space overhead.
7. **Seamless Integration with Other Algorithms:** The Z-Algorithm can be easily combined with other string processing algorithms, such as suffix arrays or hashing techniques, to enhance performance in complex DNA sequence analyses.
8. **Pattern-agnostic Efficiency:** The Z-Algorithm performs consistently well regardless of the pattern or text structure, making it a robust choice for various DNA sequence matching tasks, including those involving highly repetitive or structured sequences.

<u>Specific advantages of the Rabin-Karp Algorithm:</u>

1. **Efficient Matching with Hashing:** The Rabin-Karp algorithm efficiently matches patterns in DNA sequences by leveraging hashing, reducing the time complexity from O(m*n) in brute force to O(n) on average. This makes it highly suitable for large-scale DNA sequence analysis.

2. **Simplicity and Scalability:** The algorithm is simple to implement and scales well with increasing text size, particularly when the patterns are short relative to the text. The performance remains consistent even as the sequence length reaches billions of base pairs.
3. **Multi-pattern Matching:** Rabin-Karp can be extended to search for multiple patterns simultaneously by computing hash values for each pattern and comparing them against substrings of the text. This feature is particularly useful in genomic analysis where multiple motifs need to be identified.
4. **Hash Function Flexibility:** The choice of hash function can be tailored to specific characteristics of the DNA data, potentially improving efficiency and reducing the likelihood of hash collisions. This flexibility allows the algorithm to adapt to different types of DNA sequences.
5. **Pattern Length Impact:** The algorithm’s performance is affected by the length of the pattern, with longer patterns requiring more time for matching. However, the algorithm's efficiency improves in larger sequences, making it well-suited for handling extensive genomic data.
6. **Scalability and Performance:** The algorithm shows strong scalability, with time complexity increasing linearly with sequence length. The matching speed (Mbp/s) improves significantly with longer sequences, indicating that Rabin-Karp is particularly effective for large-scale DNA analysis.
7. **Handling Long Patterns:** The algorithm performs well with longer patterns, especially in large DNA sequences. This suggests that Rabin-Karp is efficient even when analyzing complex genomic structures, making it a versatile tool in bioinformatics.

<u>Specific advantages of KMP (Knuth-Morris-Pratt) Algorithm:</u>

1. **Linear Time Complexity:** KMP has a time complexity of $O(n + m)$, where n is the length of the text and m is the length of the pattern. This makes it efficient for both short and long patterns, ensuring that the search process is completed in linear time regardless of the pattern's position in the text.
2. **Preprocessing of the Pattern:** This algorithm preprocesses the pattern to compute a failure function (or partial match table, LPS array), which allows it to avoid unnecessary comparisons and backtracking during the search process. This preprocessing step is critical in enabling the algorithm to skip over already-matched portions of the text.
3. **Efficient for Repeated Patterns:** KMP is particularly effective when dealing with patterns that contain repeated sequences. The LPS array helps the algorithm skip over redundant checks, making it faster and more efficient in these scenarios.
4. **Simplicity:** KMP is relatively simple to implement compared to some other string-matching algorithms, making it a good choice when ease of implementation is a priority. Its straightforward approach does not require complex data structures or advanced heuristics.
5. **No Additional Memory for Skipping:** Unlike algorithms that require additional data structures to skip over sections of the text, KMP relies solely on the LPS array, which is derived from the pattern itself. This makes the algorithm more memory-efficient.
6. **Ideal for Exact Matching:** KMP is designed specifically for exact pattern matching. It is ideal for scenarios where precise matching is required, and partial or approximate matches are not acceptable.
7. **No Backtracking Required:** Once a character in the text is processed, KMP never goes back to re-check it, which makes the algorithm more efficient and faster in practice, particularly in large texts.

Specific advantages of Boyer-Moore Algorithm:

1. **Heuristic-Based Skipping:** The Boyer-Moore algorithm uses two powerful heuristics bad character rule and good suffix rule to skip large portions of the input text, making it very efficient for searching long patterns in large texts.
2. **Right-to-Left Scanning:** The algorithm scans the pattern from right to left, which often results in fewer comparisons compared to other string-matching algorithms.
3. **Best for Long Patterns:** Boyer-Moore is particularly effective for long patterns and sparse matches because it can skip sections of the text that do not contain the pattern, reducing the overall number of comparisons.
4. **Good Performance on Larger Alphabets:** The algorithm performs well on larger alphabets, making it suitable for applications like DNA sequence analysis.
5. **Low Average-Case Complexity:** While the worst-case time complexity of Boyer-Moore is O(n * m) (where n is the length of the text and m is the length of the pattern), its average-case complexity is much lower, often approaching O(n/m), especially in situations where mismatches occur early in the pattern comparison.
6. **Adaptability to Different Patterns:** Boyer-Moore is versatile and adapts well to different types of patterns, whether they are simple, complex, or repetitive. The algorithm’s heuristics dynamically adjust based on the pattern and text characteristics, optimizing the search process.
7. **Preprocessing Flexibility:** The preprocessing steps in Boyer-Moore are flexible and allow for the quick generation of the bad character and good suffix tables. This preprocessing is relatively simple and contributes significantly to the algorithm's efficiency during the search phase.

Specific advantages of Aho-Corasick Algorithm:

1. **Multiple Pattern Matching:** Aho-Corasick excels at simultaneously searching for multiple patterns in a single pass through the text, making it highly efficient for scenarios where numerous patterns need to be identified.
2. **Linear Time Complexity:** The algorithm operates in linear time relative to the size of the input text and the total length of all patterns. This ensures that the search process is efficient and scalable, even with large datasets and numerous patterns.
3. **Prefix Matching Optimization:** The algorithm efficiently handles patterns with common prefixes, avoiding repeated comparisons for shared pattern beginnings.
4. **Single-Pass Text Processing:** The text is scanned only once, regardless of the number of patterns, which is particularly beneficial for large texts like genomic sequences.
5. **Immediate Output of Matches:** The algorithm can output matches as soon as they are found, which is beneficial in real-time systems or when immediate feedback is required during the search process.
6. **Failure Function Efficiency:** The use of failure links allows quick transitions to the next potential match state, enhancing performance especially in cases of partial matches.
7. **Efficient Use of Memory:** Although it handles multiple patterns, Aho-Corasick is efficient in its use of memory. The trie structure with failure links minimizes the need for redundant data storage, making the algorithm suitable for large-scale applications.
8. **Adaptability to Streaming Data:** The algorithm can be adapted to work with streaming text, processing data as it arrives rather than requiring the entire text upfront.
9. **Versatility Across Applications:** Aho-Corasick is highly versatile and can be applied in various domains, including text searching, network intrusion detection, DNA sequence

analysis, spam filtering, and digital forensics, where multiple patterns need to be identified quickly and accurately.

10. **Minimal Overhead in Preprocessing:** The preprocessing required to build the trie and failure links is efficient and straightforward, allowing for quick setup even when dealing with a large number of patterns.
11. **Highly Scalable:** Due to its linear time complexity and ability to handle multiple patterns simultaneously, the Aho-Corasick algorithm scales well with increasing data sizes and pattern counts, making it suitable for modern applications that deal with big data.

Comparison Between Algorithm (Performances):

Algorithm	Pre-process	Avg. time	Worst time	Space	Multiple Patterns
Naïve	O(1)	O(nm)	O(nm)	O(1)	No
Rabin-Karp	O(m)	O(n+m)	O(nm)	O(1)	Yes
KMP	O(m)	O(n+m)	O(n+m)	O(m)	No
Boyer-Moore	O(m+σ)	O(n/m)	O(nm)	O(m+σ)	No
Aho-Corasick	O(M)	O(n+m+Z)	O(n+m+z)	O(M)	Yes
Ukkonen	O(n)	O(n)	O(n)	O(n)	Yes
EPMSPP	O(n+m)	O(n+m)	O(n+m)	O(n+m)	Yes
Z-Algorithm	O(m+n)	O(m+n)	O(m+n)	O(m+n)	No

Notes:

1. For Aho-Corasick, M is the sum of the lengths of all patterns.
2. For Ukkonen's algorithm, the complexities are for constructing the suffix tree. Once constructed, pattern matching is O(m+z) where z is the number of occurrences.
3. EPMSPP complexities are based on the information provided in the research paper.
4. Space complexity for EPMSPP includes the space for index tables.
5. Boyer-Moore's average case can be sublinear in practice, but this depends on the pattern and text characteristics.
6. For Z Algorithm: n is the length of the text, m is the length of the pattern, σ is the size of the alphabet, M is the sum of the lengths of all patterns, Z is the number of pattern occurrences
7. **Preprocess Time**: This refers to the time taken to prepare the pattern or input data before the actual searching begins. It's a one-time cost paid upfront.
8. **Average Case**: This represents the expected time complexity for searching based on typical input data. It's what you might experience in practice.
9. **Worst Case**: This indicates the maximum time the algorithm could take, given the worst possible input.
10. **Space**: This shows the amount of memory the algorithm needs, excluding the input data. It's important when dealing with large DNA datasets.
11. **Multiple Patterns**: Indicates whether the algorithm can handle searching for more than one pattern simultaneously, which is crucial in genomic application

Proof of Correctness for EPMSPP Algorithm:

Prerequisites:
- Let T be the text of length n
- Let P be the pattern of length m
- Let Σ be the alphabet {A, C, G, T}

Lemmas:
1. For any pair of characters $(c1, c2) \in \Sigma^2$, the index tables store all occurrences in T and P.
2. If P occurs in T, then every pair in P must occur in T at the corresponding positions.
3. The least frequent pair λ in P must occur at least once in T for any valid match.

Proof:

1. Initialization:
   - Index tables IT and IP are created for T and P respectively.
   - $\forall(c1, c2) \in \Sigma^2$, $IT(c1, c2) = \{i \mid T[i,i+1] = c1c2, 1 \le i < n\}$
   - $\forall(c1, c2) \in \Sigma^2$, $IP(c1, c2) = \{i \mid P[i,i+1] = c1c2, 1 \le i < m\}$

   Explanation: These tables store positions of all character pairs in T and P.

2. Least Frequent Pair Selection:
   - $\lambda = \text{argmin}\_\{(c1,c2) \in \Sigma^2\} \{|IT(c1, c2)| \mid IP(c1, c2) \neq \emptyset\}$

   Explanation: Find the pair that occurs least in T but exists in P.

3. Matching Process:

```
∀i ∈ IT(λ):
a. pos = i - IP(λ)[0]
b. If pos < 0 or pos + m > n, continue
c. match = true
d. ∀j ∈ [0, m-1):
   If (T[pos+j], T[pos+j+1]) ≠ (P[j], P[j+1]):
      match = false
      break
e. If match is true, report occurrence at pos
```

4. Correctness:
   - By Lemma 1 and 2, all potential match positions are considered.
   - By the matching process, all reported matches are verified.
   - By Lemma 3, no valid matches can be missed.

Therefore, EPMSPP correctly finds all occurrences of P in T.

Proof of Correctness for Ukkonen's Algorithm:

Prerequisites:
- Let S be the input string of length n
- Let ST(i) be the suffix tree for S[1..i]

Lemmas:
1. After processing character i, ST(i) represents all suffixes of S[1..i].
2. For any internal node representing αx, there exists a suffix link to a node representing x.
3. The total number of newly created nodes is O(n).
4. The time spent traversing existing nodes is amortized O(n).

Proof:

1. Invariant Maintenance:
   Base case: ST(1) is correct, containing a single edge.
   Inductive step: Assume ST(i-1) is correct. For ST(i):
   a. ∀ implicit suffixes in ST(i-1): remain unchanged in ST(i)
   b. ∀ explicit suffixes in ST(i-1):
      - If no extension needed, remain unchanged
      - If extension needed, apply Rule 1 (new leaf) or Rule 2 (split edge)
   c. New suffix S[i..i] added as per Rule 1

2. Suffix Links:
   - When a new internal node v is created for αx:
     a. If a node for αx' exists (x' being x without first character),
        set suffix link from v to node of x'
     b. Else, create node for x' in next step and link then
   Explanation: Suffix links connect related substrings for efficient future traversal.

3. Time Complexity:
   - By Lemma 3, node creation is O(n)
   - By Lemma 4 and use of suffix links, traversal is amortized O(n)
   ∴ Total time complexity is O(n)

4. Completeness:
   - By inductive proof in step 1, all suffixes are represented

5. Minimality:
   - No two edges from a node have the same initial character
   - Each internal node (except root) has at least two children
   Explanation: These conditions ensure a unique path for each suffix and a compact tree structure.

Therefore, Ukkonen's algorithm correctly constructs the minimal suffix tree for S in O(n) time.

Proof of Correctness for the Rabin-Karp Algorithm:

Prerequisites:

- Let T be the text of length n.
- Let P be the pattern of length m.
- Let Σ be the alphabet {A, C, G, T}.
- Let H(x) represent the hash of string x.

Lemmas:

1. For any string P of length m, H(P) uniquely represents P with high probability.
2. If H(P) == H(T[i+m]), then P == T[i+m] with high probability.
3. The rolling hash function correctly updates the hash value in constant time.

Proof:

Initialization:

- Compute H(P) for the pattern P.
- Compute H(T[0]) for the first substring of T of length m.

Matching Process:

For each position i in T from 0 to n-m:

a. Compare H(P) with H(T[i+m]).

b. If H(P) == H(T[i+m]), verify by comparing the actual substrings.

c. Use the rolling hash to update H(T[i+m]) to H(T[i+1+m+1]) efficiently.

Correctness:

- By Lemma 2, if H(P) == H(T[i+m]), P == T[i+m] with high probability, ensuring the match is correct.
- By Lemma 3, the rolling hash function maintains constant time complexity for each update, ensuring the overall algorithm runs in O(n) time on average.

Therefore, the Rabin-Karp algorithm correctly identifies all occurrences of P in T with high probability, making it both an effective and efficient choice for DNA sequence analysis.

Proof of Correctness for Z-Algorithm:

Proof:

1. Initialization:

- The Z-array is initialized with $Z[0]=0$, as the entire string is not considered a proper prefix of itself.
- The algorithm starts by considering the prefix of the string, initializing a left and right boundary $[L,R]$ to track the current matching segment.

2. Computation of Z-Array:

- For each position ii in the string SS:

a. **Case 1: $i>R$:**

  - L and R *are reset to* i, and a new match is attempted by comparing $S[i..n-1]$ with $S[0..n-1]$.
  - $Z[i]$ is set to the length of this match, and R is updated to the end of the match.

b. **Case 2: $i\leq R$:**

  - The algorithm uses the previously computed Z-values to potentially skip unnecessary comparisons:
    - If $Z[i-L]<R-i+1$, then $Z[i]=Z[i-L]$ (the value can be directly reused).
    - If $Z[i-L]\geq R-i+1$, a new comparison is started from $R+1$ to extend the match.

- By leveraging previously computed Z-values and only extending matches when necessary, the algorithm ensures efficient computation of the Z-array.

3. Correctness:

- **Lemma 1:** The Z-array correctly computes the length of the longest prefix match for each position i, as each Z-value is either reused from a previous calculation or determined through a direct comparison starting from $R+1$.
- **Lemma 2:** The computed Z-values accurately represent the lengths of the matching substrings, as verified through direct comparisons or reusing prior Z-values.
- **Lemma 3:** The algorithm's use of the boundaries $[L,R]$ and the reuse of Z-values ensures that each position i is processed in amortized constant time, leading to an overall time complexity of $O(n)$.

4. Time Complexity:

- The Z-Algorithm runs in $O(n)$ time for a string of length n, as each character in the string is processed at most twice: once as part of an active segment and possibly once during an extension beyond R.

5. Application to Pattern Matching:

- When used for pattern matching, the Z-Algorithm can identify all occurrences of a pattern P within a text T by constructing a concatenated string $S=P+\$+T$S, where $ is a delimiter not present in either P or T.
- By computing the Z-array for S and examining the Z-values corresponding to T, the algorithm correctly identifies all starting positions of P within T.

Proof of Correctness for KMP (Knuth-Morris-Pratt) Algorithm:

Prerequisites:
- Let T be the text of length n

- Let P be the pattern of length m
- Let lps[] be the longest proper prefix which is also suffix array for P

Lemmas:
1. The lps[] array correctly computes the longest proper prefix that is also suffix for each prefix of P.
2. When a mismatch occurs after matching j characters, the next possible match in T can start only after skipping lps[j-1] characters in P.

Proof:

1. Initialization:
   - Compute the lps[] array for pattern P. This array helps determine how much to skip in case of a mismatch.
   - Set i = 0 (index for T), and j = 0 (index for P).

2. Matching Process:

```
While i < n:
 a. If P[j] == T[i]:
     - Increment both i and j (move to the next characters in T and P).
 b. If j == m (Complete match of P in T is found):
     - Report match at index i-j
     - Set  j = lps[j-1] (to continue searching for the next possible match).
  c. Else if i < n and P[j] != T[i] (mismatch occurs) :
      If j != 0:
         - Set  j = lps[j-1] (to use the information from the prefix-suffix relationship)

      Else:
         - Increment i (move to the next character in T)
```

Correctness:

1. By Lemma 1, the lps[] array correctly identifies the longest prefix P that is also suffix.

2. By Lemma 2, when a mismatch occurs after j characters, the KMP algorithm safely skips over lps[j-1] characters in P, ensuring no potential matches are missed.
3. The algorithm examines all possible positions in T where P could match:
   - It moves the pattern based on lps[]lps[]lps[] to avoid unnecessary comparisons, ensuring an efficient search.
   - A match is reported only when all mmm characters of PPP match the corresponding characters in TTT.
4. The use of the lps[] array ensures that the KMP algorithm does not perform redundant comparisons, and it correctly handles overlapping patterns within the text.

Therefore, KMP correctly finds all occurrences of P in T.

Proof of Correctness for Boyer-Moore Algorithm:

1. Prerequisites:

T - The text of length n.
P - The pattern of length m.
R[] - The bad character heuristic array, which stores the rightmost occurrence of each character in P.
N[] - The good suffix heuristic array, which stores the appropriate shifts based on the matched suffixes.

2. Lemmas:

Lemma 1: The R[] array correctly stores the rightmost occurrence of each character in P. This allows the algorithm to skip over characters in the text that do not match the pattern's characters, ensuring efficient progression through the text.
Lemma 2: The N[] array correctly stores the shift values for the good suffix heuristic. This ensures that when a partial match is found, the pattern can be shifted in such a way that it is aligned with the next potential match in the text.
Lemma 3: Taking the maximum of the shifts provided by the bad character heuristic (R[]) and the good suffix heuristic (N[]) is always safe. This means that the Boyer-Moore algorithm will never skip over a valid match.

Proof:
Initialization:
Step 1: Compute the R[] array for the bad character heuristic. This array allows the algorithm to determine how far to shift the pattern when a mismatch occurs based on the last occurrence of the mismatched character in the pattern.
Step 2: Compute the N[] array for the good suffix heuristic. This array helps determine how far to shift the pattern when a suffix of the pattern matches a substring of the text.
Step 3: Set i = 0 , the starting index in T.

Matching Process:

```
- While i ≤ n - m:
    1. Set j = m - 1 (start comparing from the end of the pattern).
    2. While j ≥ 0 and P[j] == T[i+j]:
        - Decrement j (move left in the pattern and continue comparing).
    3.  If j < 0 (indicating a complete match of P in T):
        - Report a match at index i.
        -  Shift the pattern by N[0] positions to the right.
    4. Else (a mismatch occurred):
        - Shift the pattern by max(N[j+1], j - R[T[i+j]]) positions to the right.
```

Correctness:
1. By Lemma 1, the R[] array ensures that the pattern is shifted to align with the rightmost occurrence of the mismatched character in the pattern. If the character is not in the pattern, the pattern can be shifted past the current text character.

2. By Lemma 2, the N[] array allows the pattern to be shifted so that the next possible occurrence of the pattern's suffix in the text is aligned, avoiding unnecessary comparisons.

3. By Lemma 3, taking the maximum of the shifts provided by the bad character and good suffix heuristics ensures that the algorithm always moves the pattern to a position that does not miss any potential matches, while still maximizing the efficiency of the search.
4. The algorithm considers all possible match positions by systematically moving the pattern through the text. It ensures that every potential alignment of the pattern with the text is checked for a match, but without rechecking positions that are already known to be mismatches.
5. The algorithm reports a match only when all m characters of P match the corresponding characters in T. The use of the R[] and N[] arrays ensures that unnecessary comparisons are skipped, making the Boyer-Moore algorithm efficient.

Thus, the Boyer-Moore algorithm correctly finds all occurrences of the pattern P in the text T.

Proof of Correctness for Aho-Corasick Algorithm:

Prerequisites:

- T - The text of length n.
- P - The set of patterns P1, P2, ..., Pk, each with length mi.
- Trie - A data structure built from the patterns in P where each path from the root to a leaf node represents a pattern.
- Failure Links - Links that allow the algorithm to fall back to the longest proper suffix when a mismatch occurs during the search.
- Output Links - Links that connect nodes to other nodes representing patterns that are suffixes of the pattern at the current node.

Lemmas:

Lemma 1:

The Trie structure correctly represents all patterns in P. Each pattern in P is uniquely represented by a path from the root of the Trie to a corresponding leaf node.

Lemma 2:
Failure links correctly point to the longest proper suffix that is also a prefix of any pattern. This ensures that when a mismatch occurs, the algorithm can efficiently transition to the appropriate node in the Trie, avoiding redundant comparisons.

Lemma 3:
Output links correctly identify all patterns that are suffixes of the current pattern being processed. This ensures that all occurrences of patterns in the text are reported.

Proof:

Initialization:

1. Construct the Trie:

- Insert each pattern Pi into the Trie by sequentially adding its characters.
- Each node in the Trie corresponds to a prefix of some pattern, and the leaf nodes mark the end of a pattern.

2. Build Failure Links:
   - Use a breadth-first search (BFS) to construct failure links.
   - For each node v, set its failure link to the node corresponding to the longest suffix of the string represented by v that is also a prefix of some pattern.
   - If no such suffix exists, set the failure link to the root.

3. Construct Output Links:

   - Traverse the Trie again and for each node, if the node reached by the failure link has an output (it represents a suffix that is also a pattern), add an output link from the current node to that node.

   - This ensures that all patterns that are suffixes of the current pattern can be efficiently reported.

Matching Process:

1. Start at the root of the Trie.
2. For each character in the text T:

   a. If a transition exists for the character:

   - Move to the child node corresponding to this character.

   b. If no transition exists:

   - Follow the failure link until a valid transition is found or return to the root.

   c. Check for Output Links:

   - If the current node has an output link, report matches for all patterns linked to that node.

3. Repeat this process for the entire text to identify all occurrences of the patterns in P.

Correctness:

1. By Lemma 1, the Trie correctly represents all patterns in P. The construction of the Trie guarantees that all patterns are represented and that each pattern can be uniquely identified by a path from the root to a leaf node.

2. By Lemma 2, failure links allow the algorithm to efficiently backtrack when a mismatch occurs. This ensures that the search process is efficient and does not involve redundant comparisons. Failure links always point to the longest suffix that can still potentially match the text, minimizing unnecessary backtracking.

3. By Lemma 3, output links ensure that all patterns that are suffixes of the current pattern being processed are reported. This guarantees that the algorithm correctly identifies and reports all occurrences of patterns in P within the text T.

Thus, the Aho-Corasick algorithm correctly finds all occurrences of all patterns in P within the text T.

COMPARATIVE ANALYSIS:

EPMSPP:

- vs. Ukkonen's Algorithm:
  - EPMSPP: Faster for multiple different DNA sequences, lower memory usage
  - Ukkonen's: Extremely fast queries once built, but high initial construction cost
- vs. Boyer-Moore:
  - EPMSPP: Tailored for DNA, consistent performance across pattern lengths (29-55 Mbp/s)
  - Boyer-Moore: More versatile for general text, slightly faster for longer patterns (138-176 Mbp/s)
- vs. Aho-Corasick:
  - EPMSPP: More efficient for single pattern matching in DNA
  - Aho-Corasick: Superior for simultaneous multiple pattern matching (63-71 Mbp/s for multiple patterns)

EPMSPP excels in DNA sequence analysis with its nucleotide pair indexing approach. It offers consistent performance (29-55 Mbp/s) across various pattern lengths, outperforms general-purpose algorithms in DNA contexts, and balances speed with memory efficiency. Ideal for large-scale genomic data analysis, EPMSPP is particularly effective for frequent searches on different DNA sequences.

Ukkonen's Algorithm:

- vs. EPMSPP:
  - Ukkonen's: Extremely fast queries once built, versatile for various string operations
  - EPMSPP: Faster for multiple different sequences, lower memory usage
- vs. KMP:
  - Ukkonen's: Superior for multiple queries on the same text
  - KMP: Better for single or few queries, lower memory usage
- vs. Aho-Corasick:
  - Ukkonen's: More versatile for complex string operations
  - Aho-Corasick: Superior for multi-pattern matching

Ukkonen's algorithm provides linear-time suffix tree construction and exceptionally fast subsequent queries. It excels in scenarios requiring repeated complex string operations on a fixed text. Ideal for applications like genome browsers or DNA databases, it offers unparalleled query speed and flexibility for large, static sequences, but at the cost of higher memory usage and initial construction time.

Rabin-Karp Algorithm:

- vs. KMP:
  - Rabin-Karp: Better scalability for very large texts, good for multiple patterns

    - KMP: More efficient for single pattern, no false positives
- vs. Boyer-Moore:
    - Rabin-Karp: Simpler implementation, easily extends to multiple patterns
    - Boyer-Moore: Generally faster for single pattern matching

Rabin-Karp excels in scalability, showing impressive performance for large sequences (up to 187.79 Mbp/s for 1 billion base pairs). Its hash-based approach allows easy extension to multiple pattern matching. While prone to false positives, it's particularly effective for large-scale DNA sequence analysis and flexible pattern searching in massive datasets.

<u>KMP (Knuth-Morris-Pratt) Algorithm:</u>

- vs. Boyer-Moore:
    - KMP: More consistent performance, better worst-case scenario
    - Boyer-Moore: Often faster in practice, especially for longer patterns
- vs. Naive string search:
    - KMP: Linear time complexity, no backtracking
    - Naive: Simpler implementation, but quadratic worst-case time

KMP offers consistent performance (119-139 Mbp/s) and guaranteed linear-time complexity (O(n+m)). It excels with patterns containing repeating sub-patterns, common in DNA. KMP's no-backtracking approach makes it reliable for real-time applications.

<u>Boyer-Moore Algorithm:</u>

- vs. KMP:
    - Boyer-Moore: Often faster in practice, especially for longer patterns
    - KMP: More consistent performance, better worst-case scenario
- vs. Naive string search:
    - Boyer-Moore: Sublinear time in best case, uses text information
    - Naive: Simpler, but always compares every character

Boyer-Moore shows excellent practical performance, especially for longer patterns and larger alphabets. In DNA analysis, it achieves 138-176 Mbp/s, slightly outperforming KMP. Its efficiency increases with pattern length, making it ideal for searching long DNA motifs or gene sequences. Boyer-Moore adapts well to both pattern and text characteristics, excelling in general-purpose string matching.

<u>Aho-Corasick Algorithm:</u>

- vs. EPMSPP:
    - Aho-Corasick: Efficient for simultaneous multiple pattern matching
    - EPMSPP: More efficient for single pattern matching in DNA
- vs. Multiple runs of single-pattern algorithms:
    - Aho-Corasick: One pass for multiple patterns
    - Single-pattern algorithms: Require multiple passes

Aho-Corasick is the top choice for multi-pattern string matching, achieving 63-71 Mbp/s for multiple patterns simultaneously. It uses a trie structure with failure links, allowing efficient matching without backtracking. Aho-Corasick excels in scenarios like DNA motif searching or virus signature detection, where multiple patterns need to be identified in a single text pass.

Z-Algorithm:

- vs. KMP:
  - Z-Algorithm: Provides comprehensive prefix information
  - KMP: More straightforward for simple pattern matching

The Z-Algorithm specializes in prefix analysis, constructing a Z-array that encodes information about all string prefixes. Performance varies (1-86 Mbp/s) based on the scenario. It is particularly useful for detecting repetitive structures in DNA and analyzing string periodicities. The Z-Algorithm offers linear time complexity and is valuable for specialized bioinformatics applications involving prefix-based analyses.

Key Concepts, Insights, and Generalizations:

1. Preprocessing for Efficiency: Most advanced string-matching algorithms employ preprocessing to improve runtime efficiency. Examples include KMP's LPS array and Boyer-Moore's bad character and good suffix tables.
   Insight: Preprocessing allows algorithms to skip unnecessary comparisons during the main search, often leading to linear or sublinear time complexity.

2. Space-Time Tradeoffs: Algorithms often trade memory for speed, or vice versa. Ukkonen's Suffix Tree is a prime example, using extra space to store all suffixes for faster queries.
   Insight: The choice between space and time efficiency depends on the specific use case and available resources.

3. Sliding Window Technique: Many string matching algorithms use a sliding window approach to efficiently process the text. Rabin-Karp's rolling hash exemplifies this.
   Insight: This technique allows for updating the search state incrementally, avoiding redundant computations.

4. Failure Functions and Smart Skipping: Advanced algorithms use failure functions or smart skipping techniques to avoid unnecessary comparisons. KMP's failure function and Boyer-Moore's bad character rule are examples.
   Insight: These techniques allow algorithms to skip characters that are known to cause a mismatch, potentially leading to sublinear time complexity in practice.

5. Trie-based Structures for Multiple Patterns: For multiple pattern matching, trie-based structures prove highly efficient, as seen in the Aho-Corasick algorithm.
   Insight: This structure allows matching multiple patterns simultaneously in linear time with respect to the text length.

6. Hashing for Quick Comparisons: Hashing is used to quickly compare substrings without character-by-character matching, as in the Rabin-Karp algorithm.

Insight: By comparing hash values instead of strings, algorithms can quickly identify potential matches and only perform full comparisons when necessary.

7. Prefix-based Analysis: Some algorithms focus on analyzing prefixes to gain insights about the entire string, like the Z-Algorithm.
   Insight: This approach allows for efficient string matching and provides valuable information about string structure and repetitions.

8. Amortized Analysis in Algorithm Design: Many of these algorithms achieve their efficiency through amortized analysis, where costly operations are balanced out by cheaper ones over time.
   Insight: This technique allows algorithms like Ukkonen's to achieve linear time complexity despite some potentially costly operations.

9. DNA-Specific Optimizations: Algorithms like EPMSPP use DNA-specific optimizations to improve efficiency, such as pair indexing for the limited DNA alphabet.
   Insight: Leveraging domain-specific knowledge can lead to significant performance improvements.

10. Generalization vs. Specialization: The algorithms demonstrate a spectrum from highly specialized (like EPMSPP for DNA) to general-purpose (like KMP or Boyer-Moore).
    Insight: The choice between a specialized or general algorithm depends on the specific requirements of the problem and the nature of the data.

11. Suffix Links and Failure Transitions: Many advanced string algorithms use concepts like suffix links (in Ukkonen's algorithm) or failure transitions (in Aho-Corasick) to efficiently navigate between related states.
    Insight: These links allow for quick transitions in the data structure, reducing the overall time complexity of the algorithms.

12. Online vs. Offline Algorithms: Some algorithms, like KMP and Aho-Corasick, can process the text in a streaming fashion (online), while others, like suffix tree construction, typically require the entire text upfront (offline).
    Insight: Online algorithms are crucial for real-time processing or when dealing with very large texts that don't fit in memory.

13. Deterministic vs. Probabilistic Approaches: While most algorithms discussed are deterministic, Rabin-Karp introduces a probabilistic element with its hash function.
    Insight: Probabilistic approaches can offer efficiency gains at the cost of a small probability of error, which is often acceptable in certain applications.

14. Alphabet Size Considerations: The efficiency of many algorithms, particularly those using preprocessing (like Boyer-Moore), can be affected by the size of the alphabet.
    Insight: Algorithms can be optimized based on the known characteristics of the input alphabet, as seen in DNA-specific algorithms.

15. Pattern Length vs. Text Length: Different algorithms perform differently based on the relative lengths of the pattern and text. For instance, Boyer-Moore tends to perform better for longer patterns.
    Insight: The choice of algorithm can be optimized based on known characteristics of the

expected patterns and texts.

16. Worst-Case vs. Average-Case Performance: Algorithms like KMP offer consistent worst-case performance, while others like Boyer-Moore may have poor worst-case but excellent average-case performance.
    Insight: The choice between worst-case guarantees and average-case efficiency depends on the specific requirements of the application.

17. Extensibility to Approximate Matching: While the focus is on exact matching, some of these algorithms, particularly those based on dynamic programming or automata, can be extended to approximate string matching.
    Insight: Understanding the core principles of exact matching algorithms can provide a foundation for more complex string processing tasks.

IMPLEMENTATIONS / CODE FOR ALL ALGORITHMS:

Rabin-Karp Code:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class RabinKarp {
  private static final int d = 4;  // Number of characters in the DNA alphabet (A, C,
G, T)
  private static final int q = 101;  // A prime number, should be large enough to
reduce collision

  public static List<Integer> search(String pattern, String text) {
    int m = pattern.length();
    int n = text.length();
    int i, j;
    int p = 0;  // Hash value for pattern
    int t = 0;  // Hash value for text
    int h = 1;

    List<Integer> result = new ArrayList<>();

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < m - 1; i++) {
      h = (h * d) % q;
    }

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < m; i++) {
      p = (d * p + pattern.charAt(i)) % q;
      t = (d * t + text.charAt(i)) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= n - m; i++) {
      // Check the hash values of the current window of text and pattern
      if (p == t) {
        // If the hash values match, check the characters one by one
        for (j = 0; j < m; j++) {
```

```java
          if (text.charAt(i + j) != pattern.charAt(j)) {
            break;
          }
        }

        // If p == t and pattern[0...M-1] = text[i, i+1, ...i+M-1]
        if (j == m) {
          result.add(i);
        }
      }

      // Calculate hash value for the next window of text: Remove leading digit, add
trailing digit
      if (i < n - m) {
        t = (d * (t - text.charAt(i) * h) + text.charAt(i + m)) % q;

        // We might get a negative value of t, convert it to positive
        if (t < 0) {
          t = (t + q);
        }
      }
    }

    return result;
  }

  public static String generateRandomDNA(int length) {
    char[] bases = {'A', 'C', 'G', 'T'};
    Random rand = new Random();
    StringBuilder sb = new StringBuilder(length);
    for (int i = 0; i < length; i++) {
      sb.append(bases[rand.nextInt(bases.length)]);
    }
    return sb.toString();
  }

  public static void testPerformance() {
    int[] lengths = {100, 1000, 10000, 100000, 1000000, 10000000, 100000000,
1000000000};
    String[] patterns = {"ACG", "ACGTACGT", "ACGTACGTACGTACGT"};  // Example patterns
of different lengths

    for (String pattern : patterns) {
      System.out.println("Testing for pattern: " + pattern);
      for (int length : lengths) {
        String text = generateRandomDNA(length);
        long startTime = System.nanoTime();
        List<Integer> matches = search(pattern, text);
        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        double matchingTimeSeconds = duration / 1e9;
        double matchingSpeedMbp = (length / 1e6) / matchingTimeSeconds;
        System.out.println("Text length: " + length);
        System.out.println("Number of matches: " + matches.size());
        System.out.println("Matching time: " + matchingTimeSeconds + " seconds");
        System.out.println("Matching speed: " + matchingSpeedMbp + " Mbp/s");
        System.out.println();
      }
    }
  }
  public static void main(String[] args) {
    testPerformance();
  }
}
```

EPMSPP ALGORITHM:

```
import java.util.*;

/*
* EPMSPPAlgorithm class implements the Exact Multiple Pattern Matching Algorithm
* using DNA Sequence and Pattern Pair (EPMSPP). This algorithm is designed to
efficiently search
* for DNA patterns in large sequences.
*/

public class EPMSPPAlgorithm {
  private static final int ALPHABET_SIZE = 16; // 4^2 for DNA pairs

  /*
   * Main method to find matches of a pattern in a sequence; Implements the Exact
Multiple
   * Pattern Matching Algorithm using DNA Sequence and Pattern Pair (EPMSPP).
   * This algorithm is designed to efficiently search for DNA patterns in large
sequences
   */
  public static List<Integer> findMatches(String sequence, String pattern) {
    List<Integer> matches = new ArrayList<>();

    // Return empty list if pattern is longer than sequence.
    if (pattern.length() > sequence.length()) return matches;

    // Create index tables for both sequence and pattern
    // These tables store the positions of each DNA pair, reducing unnecessary
comparisons.
    int[][] stab = createIndexTable(sequence);
    int[][] ptab = createIndexTable(pattern);

    // Find the least frequent pair in the pattern
    // This is a key optimization of EPMSPP, as it minimizes the number of potential
match positions
    int leastFrequentPair = findLeastFrequentPair(ptab, stab);
    int leastFrequentPairPos = ptab[leastFrequentPair][0];

    // Iterate through potential match positions
    // We only check positions where the least frequent pair occurs, greatly reducing
comparisons
    for (int i = 0; i < stab[leastFrequentPair].length && stab[leastFrequentPair][i]
!= -1; i++) {
      int potentialMatchPos = stab[leastFrequentPair][i] - leastFrequentPairPos;

      // change1: if (potentialMatchPos < 0) continue;
      if (potentialMatchPos < 0 || potentialMatchPos + pattern.length() >
sequence.length()) continue;

      // Check if the potential match is a full match.
      // We compare pairs of characters, which is more efficient than character-by-
character comparison
```

```java
      boolean match = true;

      for (int j = 0; j < pattern.length() - 1; j++) {
        int pairIndex = computePairIndex(pattern.charAt(j), pattern.charAt(j + 1));
        int seqPairIndex = computePairIndex(sequence.charAt(potentialMatchPos + j),
                sequence.charAt(potentialMatchPos + j + 1));
        if (pairIndex != seqPairIndex) { match = false; break; }
      }


      if (match) {  matches.add(potentialMatchPos); }
    }

    return matches;
  }




  /*
   * Creates an index table for a given string
   * This is a key component of the EPMSPP algorithm, reducing the search space
   * The table stores positions of each DNA pair in the string
   */
  private static int[][] createIndexTable(String str) {
    int[][] table = new int[ALPHABET_SIZE][str.length()];
    for (int[] row : table) Arrays.fill(row, -1);

    int[] counts = new int[ALPHABET_SIZE];

    // Populate the table with positions of each DNA pair
    // This allows for quick lookup of pair positions later
    for (int i = 0; i < str.length() - 1; i++) {
      int index = computePairIndex(str.charAt(i), str.charAt(i + 1));
      table[index][counts[index]++] = i;
    }
    return table;
  }




  /*
   * Computes the index for a pair of DNA bases
   * Uses bit manipulation for efficiency, as suggested in the research paper

   * For example:
   * For the pair 'AC': A (00) << 2 gives 0000, C (01) gives 01, Res: 0001 (1 in
decimal)
   * For the pair 'GT': G (10) << 2 gives 1000, T (11) gives 11, Res: 1011 (11 in
decimal)   */
  private static int computePairIndex(char c1, char c2) {

    // charToIndex(c1) & 3: This ensures we only get the last 2 bits (3 in binary is
11).
    // (charToIndex(c1) & 3) << 2: This shifts the bits to the left by 2 positions.
    // OR operation |: Combines the bits of the two characters to form a unique
index.
    return ((charToIndex(c1) & 3) << 2) | (charToIndex(c2) & 3);
  }


  /*
```

```java
   * Converts a DNA base character to its corresponding index
   * This mapping is crucial for the pair indexing system
   */
  private static int charToIndex(char c) {
    switch (c) {
      case 'A': return 0;
      case 'C': return 1;
      case 'G': return 2;
      case 'T': return 3;
      default: throw new IllegalArgumentException("Invalid DNA base: " + c);
    }
  }

  /*
   * Finds the least frequent pair in the pattern
   * This is a key optimization in EPMSPP, significantly reducing the number of
comparisons
   */
  private static int findLeastFrequentPair(int[][] ptab, int[][] stab) {
    int leastFrequent = -1;
    int minCount = Integer.MAX_VALUE;

    for (int i = 0; i < ALPHABET_SIZE; i++) {
      int pCount = 0;
      int sCount = 0;
      while (pCount < ptab[i].length && ptab[i][pCount] != -1) pCount++;
      while (sCount < stab[i].length && stab[i][sCount] != -1) sCount++;

      if (pCount > 0 && sCount < minCount) {
        minCount = sCount;
        leastFrequent = i;
      }
    }

    return leastFrequent;
  }

  /* -------------------------------------------------------------------------- */
  /* -------------------------------------------------------------------------- */
  private static class DNAGenerator {
    private static final String DNA_BASES = "ACGT";
    private Random random;

    public DNAGenerator() {
      this.random = new Random();
    }

    public String generateSequence(int length) {
      StringBuilder sequence = new StringBuilder(length);
      for (int i = 0; i < length; i++) {
        sequence.append(DNA_BASES.charAt(random.nextInt(DNA_BASES.length())));
      }
      return sequence.toString();
    }

    public String generatePattern(int length) {
      return generateSequence(length);
    }
  }

  /* -------------------------------------------------------------------------- */
  // New helper methods: for displaying context around random matches.

  private static void printRandomMatchContext(String sequence, String pattern,
List<Integer> matches) {
    if (matches.size() < 1) {
      System.out.println("Not enough matches to show context.");
      return;
    }

    List<Integer> selectedPositions = selectRandomPositions(matches, 2);
    for (int position : selectedPositions) {
      String context = extractContext(sequence, position, pattern.length(), 10);
      System.out.println("Context at position " + position + ": " + context);
    }
  }

  private static List<Integer> selectRandomPositions(List<Integer> positions, int
count) {

    if (positions.size() <= count) { return new ArrayList<>(positions); }

    List<Integer> copy = new ArrayList<>(positions);
    Collections.shuffle(copy);
    return copy.subList(0, count);
  }


  private static String extractContext(String sequence, int position, int
patternLength, int contextSize) {
    int start = Math.max(0, position - contextSize);
    int end = Math.min(sequence.length(), position + patternLength + contextSize);

    StringBuilder sb = new StringBuilder();

    // start with ellipsis if we didn't start at the beginning of the sequence.
    if (start > 0) sb.append("...");

    // start and end are the indices of the substring to extract.
    sb.append(sequence, start, position);
    // encase the pattern in square brackets.
    sb.append("["); sb.append(sequence, position, position + patternLength);
sb.append("]");
    sb.append(sequence, position + patternLength, end);

    // if we didn't reach the end of the sequence, add ellipsis.
    if (end < sequence.length()) sb.append("...");

    return sb.toString();
  }


  /* -------------------------------------------------------------------------- */


  public static void main(String[] args) {
    DNAGenerator generator = new DNAGenerator();
    long startTime, endTime, totalStartTime, totalEndTime;

    // Time the sequence generation
    startTime = System.nanoTime();
    String sequence = generator.generateSequence(100000000); //  100 million bases
    endTime = System.nanoTime();
    System.out.println("Sequence generation time: " + (endTime - startTime) / 1e9 + "
seconds");
    System.out.println("Sequence length: " + sequence.length());

    List<String> patterns = new ArrayList<>();
    patterns.add(sequence.substring(3000, 3010)); // 10-character pattern
    patterns.add(generator.generatePattern(12));  // Random 12-character pattern
    patterns.add(generator.generatePattern(14));  // Random 14-character pattern

    totalStartTime = System.nanoTime();
    double totalMatchingTime = 0;

    for (String pattern : patterns) {
      System.out.println("\nPattern: " + pattern);

      // Time the pattern matching
      startTime = System.nanoTime();
      List<Integer> matches = findMatches(sequence, pattern);
      endTime = System.nanoTime();

      // Calculate and print matching time:
      double matchingTime = (endTime - startTime) / 1e9;
      totalMatchingTime += matchingTime;

      System.out.println("Matching time: " + String.format("%.6f", matchingTime) + "
seconds");
      System.out.println("Matches found at positions: " + matches);
      System.out.println("Number of matches: " + matches.size());

      // Calculate and print matching speed
      double matchingSpeed = sequence.length() / (matchingTime * 1e6); // Million
bases per second
      System.out.println("Matching speed: " + String.format("%.2f", matchingSpeed) + "
Mbp/s");

      // Print context for two random matches
      printRandomMatchContext(sequence, pattern, matches);
    }

    totalEndTime = System.nanoTime();
    double totalTime = (totalEndTime - totalStartTime) / 1e9;

    System.out.println("\nTotal Statistics:");
    System.out.println("Total time taken for all patterns: " + String.format("%.6f",
totalTime) + " seconds");
    System.out.println("Sum of individual matching times: " + String.format("%.6f",
totalMatchingTime) + " seconds");
    System.out.println("Average matching time per pattern: " + String.format("%.6f",
totalMatchingTime / patterns.size()) + " seconds");
    System.out.println("Average matching speed: " + String.format("%.2f",
(sequence.length() * patterns.size()) / (totalMatchingTime * 1e6)) + " Mbp/s");
  }


}


/* Notes:
  * Mbp/s stands for "Million base pairs per second.
  * In genomics, a base pair is a unit of DNA (A-T or C-G). For single-stranded DNA or
```

```
RNA, we often
  just say "bases." This metric tells us how many million DNA bases the algorithm can
process in
  one second.
  * By dividing by 1e9, we convert nanoseconds to seconds.
  * */
```

UKKONEN'S ALGORITHM:

```
import java.util.*;

public class UkkonenAlgorithm {
  private static final int ALPHABET_SIZE = 4; // For DNA: A, C, G, T

  /*
   * Node class represents both internal nodes and leaves in the suffix tree.
   * It's a crucial component in Ukkonen's algorithm, allowing for efficient
   * tree construction and traversal.
   */
  static class Node {
    Node[] children;      // Array of child nodes, one for each possible DNA base
    int start;            // Start index of the substring this edge represents
    Integer end;          // End index of the substring (null for leaves)
    Node suffixLink;      // Suffix link to another node (crucial for Ukkonen's
algorithm)

    Node(int start, Integer end) {
      this.children = new Node[ALPHABET_SIZE];
      this.start = start;
      this.end = end;
      this.suffixLink = null;
    }

    // Calculate the length of the edge leading to this node
    int edgeLength(int position) {
      return end == null ? position - start + 1 : end - start;
    }
  }

  /*
   * SuffixTree class implements Ukkonen's algorithm for linear-time suffix tree
construction.
   * This data structure allows for efficient pattern matching in strings,
particularly useful
   * for DNA sequence analysis.
   */
  static class SuffixTree {
    byte[] text;            // The input text (DNA sequence) stored as bytes for
efficiency
    Node root;              // Root node of the suffix tree
    Node activeNode;        // Part of the "active point" in Ukkonen's algorithm
    int activeEdge = -1;    // Part of the "active point" in Ukkonen's algorithm
    int activeLength = 0;   // Part of the "active point" in Ukkonen's algorithm
    int remainingSuffixCount = 0;  // Number of suffixes yet to be added in current
phase
    int leafEnd = -1;       // End index for leaf nodes
    int position = -1;      // Current position in the text being processed
    List<Node> nodes;       // List of all nodes, useful for operations on the entire
tree

    /*
     * Constructor: Initializes the suffix tree and builds it using Ukkonen's
```

```
algorithm.
     * Time Complexity: O(n) where n is the length of the input string.
     * Space Complexity: O(n) in the worst case.
     */
    SuffixTree(String input) {
      // Convert input string to byte array for memory efficiency
      text = new byte[input.length() + 1];
      for (int i = 0; i < input.length(); i++) {
        text[i] = charToByte(input.charAt(i));
      }
      text[input.length()] = 0; // Terminator character

      nodes = new ArrayList<>();
      root = new Node(-1, -1);
      nodes.add(root);
      activeNode = root;

      // Build the suffix tree
      for (int i = 0; i < text.length; i++) {
        extendSuffixTree(i);
      }
    }

    // Convert DNA character to byte representation
    private byte charToByte(char c) {
      switch (c) {
        case 'A': return 0;
        case 'C': return 1;
        case 'G': return 2;
        case 'T': return 3;
        default: throw new IllegalArgumentException("Invalid DNA base: " + c);
      }
    }

    // Convert byte representation back to DNA character
    private char byteToChar(byte b) {
      switch (b) {
        case 0: return 'A';
        case 1: return 'C';
        case 2: return 'G';
        case 3: return 'T';
        default: return '$'; // Terminator character
      }
    }

    /*
     * Core method of Ukkonen's algorithm: extends the suffix tree by one character.
     * This method implements the heart of the linear-time construction process.
     * Time Complexity: Amortized O(1) per character, leading to O(n) overall.
     */
    void extendSuffixTree(int pos) {
      leafEnd = pos;
      remainingSuffixCount++;
      int lastNewNodeIndex = -1;

      while (remainingSuffixCount > 0) {
        if (activeLength == 0) activeEdge = pos;

        if (activeNode.children[text[activeEdge]] == null) {
          // Create a new leaf node
          activeNode.children[text[activeEdge]] = new Node(pos, null);
          nodes.add(activeNode.children[text[activeEdge]]);
          if (lastNewNodeIndex != -1) {
            nodes.get(lastNewNodeIndex).suffixLink = activeNode;
            lastNewNodeIndex = -1;
          }
        } else {
          Node next = activeNode.children[text[activeEdge]];
          if (walkDown(next)) continue;

          if (text[next.start + activeLength] == text[pos]) {
            activeLength++;
            if (lastNewNodeIndex != -1) {
              nodes.get(lastNewNodeIndex).suffixLink = activeNode;
            }
            break;
          }

          // Split the edge
          Node split = new Node(next.start, next.start + activeLength);
          nodes.add(split);
          activeNode.children[text[activeEdge]] = split;
          split.children[text[pos]] = new Node(pos, null);
          nodes.add(split.children[text[pos]]);
          next.start += activeLength;
          split.children[text[next.start]] = next;

          if (lastNewNodeIndex != -1) {
            nodes.get(lastNewNodeIndex).suffixLink = split;
          }
          lastNewNodeIndex = nodes.indexOf(split);
        }

        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) {
          activeLength--;
          activeEdge = pos - remainingSuffixCount + 1;
        } else if (activeNode != root) {
          activeNode = activeNode.suffixLink != null ? activeNode.suffixLink : root;
        }
      }
    }

    /*
     * Helper method for the "walk down" procedure in Ukkonen's algorithm.
     * This method is crucial for maintaining the correct active point during tree
extension.
     */
    boolean walkDown(Node node) {
      int edgeLength = edgeLength(node);
      if (activeLength >= edgeLength) {
        activeEdge += edgeLength;
        activeLength -= edgeLength;
        activeNode = node;
        return true;
      }
      return false;
    }

    // Calculate the length of an edge
    int edgeLength(Node node) {
      return node.end == null ? leafEnd - node.start + 1 : node.end - node.start;
    }
```

```java
    /*
     * Pattern matching method: finds all occurrences of a pattern in the text.
     * This showcases the power of suffix trees for efficient string matching.
     * Time Complexity: O(m + k), where m is pattern length and k is number of
occurrences.
     */
    List<Integer> findPattern(String pattern) {
      List<Integer> positions = new ArrayList<>();
      Node node = root;
      int i = 0;
      while (i < pattern.length()) {
        byte c = charToByte(pattern.charAt(i));
        if (node.children[c] != null) {
          node = node.children[c];
          int j = 0;
          while (j < edgeLength(node) && i < pattern.length() &&
                  text[node.start + j] == charToByte(pattern.charAt(i))) {
            i++;
            j++;
          }
          if (j == edgeLength(node)) {
            if (i == pattern.length()) {
              collectLeafPositions(node, positions);
              break;
            }
          } else if (i == pattern.length()) {
            collectLeafPositions(node, positions);
            break;
          } else {
            break;
          }
        } else {
          break;
        }
      }
      return positions;
    }

    /*
     * Recursive method to collect all leaf positions under a given node.
     * This is used to find all occurrences of a pattern in the text.
     */
    void collectLeafPositions(Node node, List<Integer> positions) {
      if (isLeaf(node)) {
        positions.add(text.length - node.start - 1);
      } else {
        for (Node child : node.children) {
          if (child != null) {
            collectLeafPositions(child, positions);
          }
        }
      }
    }

    // Check if a node is a leaf node
    boolean isLeaf(Node node) {
      return node.end == null;
    }
  }

  // DNA sequence generator class (unchanged)
  private static class DNAGenerator {
```

```
  private static final String DNA_BASES = "ACGT";
  private Random random;

  public DNAGenerator() {
   this.random = new Random();
  }

  public String generateSequence(int length) {
   StringBuilder sequence = new StringBuilder(length);
   for (int i = 0; i < length; i++) {
    sequence.append(DNA_BASES.charAt(random.nextInt(DNA_BASES.length())));
   }
   return sequence.toString();
  }

  public String generatePattern(int length) {
   return generateSequence(length);
  }
 }

 // Utility methods for printing match contexts (unchanged)
 private static void printRandomMatchContext(String sequence, String pattern,
List<Integer> matches) {
  if (matches.isEmpty()) {
   System.out.println("No matches found to show context.");
   return;
  }

  List<Integer> selectedPositions = selectRandomPositions(matches, Math.min(2,
matches.size()));
  for (int position : selectedPositions) {
   String context = extractContext(sequence, position, pattern.length(), 10);
   System.out.println("Context at position " + position + ": " + context);
  }
 }

 private static List<Integer> selectRandomPositions(List<Integer> positions, int
count) {
  if (positions.size() <= count) {
   return new ArrayList<>(positions);
  }

  List<Integer> copy = new ArrayList<>(positions);
  Collections.shuffle(copy);
  return copy.subList(0, count);
 }

 private static String extractContext(String sequence, int position, int
patternLength, int contextSize) {
  int start = Math.max(0, position - contextSize);
  int end = Math.min(sequence.length(), position + patternLength + contextSize);

  StringBuilder sb = new StringBuilder();
  if (start > 0) sb.append("...");
  sb.append(sequence, start, position);
  sb.append("[");
  sb.append(sequence, position, Math.min(position + patternLength,
sequence.length()));
  sb.append("]");
  sb.append(sequence, Math.min(position + patternLength, sequence.length()), end);
  if (end < sequence.length()) sb.append("...");
```

```java
    return sb.toString();
  }

  /*
   * Main method to demonstrate the Ukkonen's algorithm for DNA sequence matching.
   * It generates a DNA sequence, constructs a suffix tree, and performs pattern
matching.
   */
  public static void main(String[] args) {
    DNAGenerator generator = new DNAGenerator();
    long startTime, endTime, totalStartTime, totalEndTime;

    // Generate a DNA sequence
    startTime = System.nanoTime();
    String sequence = generator.generateSequence(1000000); // 1 million bases
    endTime = System.nanoTime();
    System.out.println("Sequence generation time: " + (endTime - startTime) / 1e9 + "
seconds");
    System.out.println("Sequence length: " + sequence.length());

    // Define patterns to search for
    List<String> patterns = new ArrayList<>();
    patterns.add(sequence.substring(3000, 3010)); // 10-character pattern from the
sequence
    patterns.add(generator.generatePattern(12));  // Random 12-character pattern
    patterns.add(generator.generatePattern(14));  // Random 14-character pattern

    totalStartTime = System.nanoTime();

    // Construct the suffix tree
    startTime = System.nanoTime();
    SuffixTree tree = new SuffixTree(sequence);
    endTime = System.nanoTime();
    System.out.println("Suffix tree construction time: " + (endTime - startTime) / 1e9
+ " seconds");

    double totalMatchingTime = 0;

    // Perform pattern matching for each pattern
    for (String pattern : patterns) {
      System.out.println("\nPattern: " + pattern);

      startTime = System.nanoTime();
      List<Integer> matches = tree.findPattern(pattern);
      endTime = System.nanoTime();

      double matchingTime = (endTime - startTime) / 1e9;
      totalMatchingTime += matchingTime;

      System.out.println("Matching time: " + String.format("%.6f", matchingTime) + "
seconds");
      System.out.println("Matches found at positions: " + matches);
      System.out.println("Number of matches: " + matches.size());

      double matchingSpeed = sequence.length() / (matchingTime * 1e6); // Million
bases per second
      System.out.println("Matching speed: " + String.format("%.2f", matchingSpeed) + "
Mbp/s");

      printRandomMatchContext(sequence, pattern, matches);
    }
    totalEndTime = System.nanoTime();
    double totalTime = (totalEndTime - totalStartTime) / 1e9;

    // Print overall statistics
    System.out.println("\nTotal Statistics:");
    System.out.println("Total time taken (including tree construction): " +
String.format("%.6f", totalTime) + " seconds");
    System.out.println("Total matching time: " + String.format("%.6f",
totalMatchingTime) + " seconds");
    System.out.println("Average matching time per pattern: " + String.format("%.6f",
totalMatchingTime / patterns.size()) + " seconds");
    System.out.println("Average matching speed: " + String.format("%.2f",
(sequence.length() * patterns.size()) / (totalMatchingTime * 1e6)) + " Mbp/s");
  }
}
```

Z-Algorithm :

```
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;
public class DNASequenceMatcherZ {
  private static final String DNA_BASES = "ACGT";
  private static final int MAX_SEQUENCE_LENGTH = 1_000_000_0; // 1 billion

  public static List<Integer> findPatternOccurrences(String sequence, String pattern)
{
    List<Integer> occurrences = new ArrayList<>();
    String combined = pattern + "$" + sequence;
    int[] z = calculateZ(combined);
    int patternLength = pattern.length();

    for (int i = patternLength + 1; i < z.length; i++) {
      if (z[i] == patternLength) {
        occurrences.add(i - patternLength - 1);
      }
    }

    return occurrences;
  }

  private static int[] calculateZ(String s) {
    int n = s.length();
    int[] z = new int[n];
    int l = 0, r = 0;
    for (int i = 1; i < n; i++) {
      if (i <= r) {
        z[i] = Math.min(r - i + 1, z[i - l]);
      }
      while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
        z[i]++;
      }
      if (i + z[i] - 1 > r) {
        l = i;
        r = i + z[i] - 1;
      }
    }
    return z;
  }

  public static String generateLargeDNASequence(int length) {
    if (length > MAX_SEQUENCE_LENGTH) {
      throw new IllegalArgumentException("Sequence length cannot exceed " +
MAX_SEQUENCE_LENGTH);
```

```
    }

    StringBuilder sequence = new StringBuilder(length);
    ThreadLocalRandom random = ThreadLocalRandom.current();

    for (int i = 0; i < length; i++) {
      sequence.append(DNA_BASES.charAt(random.nextInt(DNA_BASES.length())));
    }

    return sequence.toString();
  }

  public static void main(String[] args) {
    int sequenceLength = 1_000_000_0; // 1 billion characters
    String pattern = "ACGTACGT"; // Example pattern

    System.out.println("Generating DNA sequence of length " + sequenceLength +
"...");
    long startTime = System.currentTimeMillis();
    String dnaSequence = generateLargeDNASequence(sequenceLength);
    long endTime = System.currentTimeMillis();
    System.out.println("Generation time: " + (endTime - startTime) + " ms");

    System.out.println("Searching for pattern: " + pattern);
    startTime = System.currentTimeMillis();
    List<Integer> occurrences = findPatternOccurrences(dnaSequence, pattern);
    endTime = System.currentTimeMillis();

    System.out.println("Search time: " + (endTime - startTime) + " ms");
    System.out.println("Number of occurrences: " + occurrences.size());
    System.out.println("First 10 occurrences: " + occurrences.subList(0, Math.min(10,
            occurrences.size())));
  }
}
```

<u>KMP, Boyer Moore and Aho Corasick Algorithms:</u>

```
import java.util.*;


public class DNASequenceAnalysis {
 private static final String DNA_BASES = "ACGT";
 private static final Random random = new Random();
 private static final int CHUNK_SIZE = 10_000_000; // 10 million characters/chunk

 public static void main(String[] args) {
   long sequenceLength = 100_000_000;
   int patternLength = 10;
   int numberOfPatterns = 3;
   int numberOfTests = 3;

   for (int test = 1; test <= numberOfTests; test++) {
     // Generates the sequence
     long startTime = System.nanoTime();
     DNASequenceStream dnaStream = new DNASequenceStream(sequenceLength);
     long endTime = System.nanoTime();
     double sequenceGenerationTime = (endTime - startTime) / 1e9;
     // Generates random patterns
     String[] patterns = generateRandomPatterns(patternLength, numberOfPatterns);
     // KMP Algorithm
     long kmpStartTime = System.nanoTime();
     Map<String, Double> kmpResults = runKMPTests(dnaStream, patterns);
     long kmpEndTime = System.nanoTime();
     double kmpTotalExecutionTime = (kmpEndTime - kmpStartTime) / 1e9;
     kmpResults.put("TotalExecutionTime", kmpTotalExecutionTime);

     // Boyer-Moore Algorithm
     long bmStartTime = System.nanoTime();
     Map<String, Double> bmResults = runBoyerMooreTests(dnaStream, patterns);
     long bmEndTime = System.nanoTime();
     double bmTotalExecutionTime = (bmEndTime - bmStartTime) / 1e9;
     bmResults.put("TotalExecutionTime", bmTotalExecutionTime);

     // Aho-Corasick Algorithm
     long acStartTime = System.nanoTime();
     Map<String, Double> acResults = runAhoCorasickTests(dnaStream, patterns);
     long acEndTime = System.nanoTime();
     double acTotalExecutionTime = (acEndTime - acStartTime) / 1e9;
     acResults.put("TotalExecutionTime", acTotalExecutionTime);

     printResultTable(test, sequenceLength, sequenceGenerationTime, patterns,
kmpResults, bmResults, acResults);
   }
 }

 private static Map<String, Double> runKMPTests(DNASequenceStream dnaStream, String[]
patterns) {
   Map<String, Double> results = new HashMap<>();
   double totalMatchingTime = 0;
   double totalMatches = 0;

   for (int i = 0; i < patterns.length; i++) {
     long startTime = System.nanoTime();
     long matches = searchPatternStreamKMP(dnaStream, patterns[i]);
     long endTime = System.nanoTime();
     double matchingTime = (endTime - startTime) / 1e9;

     results.put("Pattern" + (i + 1) + "Time", matchingTime);
     results.put("Pattern" + (i + 1) + "Matches", (double) matches);
     results.put("Pattern" + (i + 1) + "Speed", dnaStream.length / matchingTime /
1e6);

     totalMatchingTime += matchingTime;
     totalMatches += matches;
   }

   results.put("TotalPatternMatchingTime", totalMatchingTime);
   results.put("AverageMatchingTimePerPattern", totalMatchingTime / patterns.length);
   results.put("AverageMatchingSpeed", (dnaStream.length * patterns.length) /
totalMatchingTime / 1e6);

   return results;
 }

 private static Map<String, Double> runBoyerMooreTests(DNASequenceStream dnaStream,
String[] patterns) {
   Map<String, Double> results = new HashMap<>();
   double totalMatchingTime = 0;
   double totalMatches = 0;
   for (int i = 0; i < patterns.length; i++) {
     long startTime = System.nanoTime();
     long matches = searchPatternStreamBM(dnaStream, patterns[i]);
     long endTime = System.nanoTime();
     double matchingTime = (endTime - startTime) / 1e9;

     results.put("Pattern" + (i + 1) + "Time", matchingTime);
     results.put("Pattern" + (i + 1) + "Matches", (double) matches);
     results.put("Pattern" + (i + 1) + "Speed", dnaStream.length / matchingTime /
1e6);

     totalMatchingTime += matchingTime;
     totalMatches += matches;
   }

   results.put("TotalPatternMatchingTime", totalMatchingTime);
   results.put("AverageMatchingTimePerPattern", totalMatchingTime / patterns.length);
   results.put("AverageMatchingSpeed", (dnaStream.length * patterns.length) /
totalMatchingTime / 1e6);

   return results;
 }

 private static Map<String, Double> runAhoCorasickTests(DNASequenceStream dnaStream,
String[] patterns) {
   Map<String, Double> results = new HashMap<>();
   AhoCorasickNode root = buildAhoCorasickTrie(patterns);
   long startTime = System.nanoTime();
   long matches = searchPatternsStreamAC(dnaStream, patterns);
   long endTime = System.nanoTime();
   double matchingTime = (endTime - startTime) / 1e9;

   results.put("TotalMatches", (double) matches);
   results.put("TotalPatternMatchingTime", matchingTime);
   results.put("Speed", dnaStream.length / matchingTime / 1e6);
   results.put("TrieSize", (double) countTrieNodes(root));

   return results;
 }

 private static void printResultTable(int testNumber, long sequenceLength, double
sequenceGenerationTime,
                                      String[] patterns, Map<String, Double>
kmpResults,
                                      Map<String, Double> bmResults, Map<String,
Double> acResults) {
   System.out.println("Test " + testNumber + " Results");
   System.out.println("Sequence Length: " + sequenceLength);
   System.out.printf("Sequence Generation Time: %.6f s%n", sequenceGenerationTime);
   System.out.println();

   printAlgorithmResults("KMP Algorithm", patterns, kmpResults);
   printAlgorithmResults("Boyer-Moore Algorithm", patterns, bmResults);
   printAlgorithmResults("Aho-Corasick Algorithm", patterns, acResults);
 }

 private static void printAlgorithmResults(String algorithmName, String[] patterns,
Map<String, Double> results) {
   System.out.println(algorithmName + " Results:");
   if (algorithmName.equals("Aho-Corasick Algorithm")) {
     // Displays the patterns and trie structure
     System.out.println("Searching for multiple patterns: " +
Arrays.toString(patterns));
     System.out.println("Aho-Corasick Trie Structure:");
     AhoCorasickNode root = buildAhoCorasickTrie(patterns);
     printTrie(root, "", "");
     System.out.println();

     // Displays the performance results
     System.out.printf("Total Matches: %.0f%n", results.get("TotalMatches"));
     System.out.printf("Total Pattern Matching Time: %.6f s%n",
results.get("TotalPatternMatchingTime"));
     System.out.printf("Matching Speed: %.2f Mbp/s%n", results.get("Speed"));
     System.out.printf("Trie size: %.0f%n", results.get("TrieSize"));
   } else {
     for (int i = 0; i < patterns.length; i++) {
       System.out.println("Pattern " + (i + 1) + ": " + patterns[i]);
       System.out.printf("Matches: %.0f%n", results.get("Pattern" + (i + 1) +
"Matches"));
       System.out.printf("Matching Time: %.6f s%n", results.get("Pattern" + (i + 1) +
"Time"));
       System.out.printf("Matching Speed: %.2f Mbp/s%n", results.get("Pattern" + (i +
1) + "Speed"));
       System.out.println();
     }
     System.out.printf("Total Pattern Matching Time: %.6f s%n",
results.get("TotalPatternMatchingTime"));
     System.out.printf("Average Matching Time per Pattern: %.6f s%n",
results.get("AverageMatchingTimePerPattern"));
     System.out.printf("Average Matching Speed: %.2f Mbp/s%n",
results.get("AverageMatchingSpeed"));
   }
   System.out.printf("Total Execution Time: %.6f s%n",
results.get("TotalExecutionTime"));
   System.out.println();
 }

 // KMP Algorithm
 public static List<Integer> kmpSearch(String text, String pattern) {
   List<Integer> matches = new ArrayList<>();
   int[] lps = computeLPSArray(pattern);
   int i = 0, j = 0;
   while (i < text.length()) {
     if (pattern.charAt(j) == text.charAt(i)) {
       j++;
       i++;
     }
     if (j == pattern.length()) {
       matches.add(i - j);
       j = lps[j - 1];
     } else if (i < text.length() && pattern.charAt(j) != text.charAt(i)) {
       if (j != 0) {
         j = lps[j - 1];
       } else {
         i++;
       }
     }
   }
   return matches;
 }

 private static int[] computeLPSArray(String pattern) {
   int[] lps = new int[pattern.length()];
   int len = 0;
```

```java
  int i = 1;

  while (i < pattern.length()) {
    if (pattern.charAt(i) == pattern.charAt(len)) {
      len++;
      lps[i] = len;
      i++;
    } else {
      if (len != 0) {
        len = lps[len - 1];
      } else {
        lps[i] = 0;
        i++;
      }
    }
  }
  return lps;
}

// Optimized Boyer-Moore Algorithm with strong good suffix rule
public static List<Integer> boyerMooreSearch(String text, String pattern) {
  List<Integer> matches = new ArrayList<>();
  int[] badChar = buildBadCharTable(pattern);
  int[] goodSuffix = buildGoodSuffixTable(pattern);

  int n = text.length();
  int m = pattern.length();
  int i = 0;

  while (i <= n - m) {
    int j = m - 1;

    while (j >= 0 && pattern.charAt(j) == text.charAt(i + j)) {
      j--;
    }

    if (j < 0) {
      matches.add(i);
      i += (i + m < n) ? m - badChar[text.charAt(i + m)] : 1;
    } else {
      int badCharShift = j - badChar[text.charAt(i + j)];
      int goodSuffixShift = 0;

      if (j < m - 1) {
        goodSuffixShift = goodSuffix[j + 1];
      }

      i += Math.max(badCharShift, goodSuffixShift);
    }
  }

  return matches;
}

private static int[] buildBadCharTable(String pattern) {
  int[] badChar = new int[256];
  Arrays.fill(badChar, -1);

  for (int i = 0; i < pattern.length(); i++) {
    badChar[pattern.charAt(i)] = i;
  }
```

```
    return badChar;
}

private static int[] buildGoodSuffixTable(String pattern) {
    int m = pattern.length();
    int[] goodSuffix = new int[m];
    int[] suffixes = computeSuffixes(pattern);
    Arrays.fill(goodSuffix, m);
    for (int i = m - 1, j = 0; i >= 0; i--) {
        if (suffixes[i] == i + 1) {
            for (; j < m - 1 - i; j++) {
                if (goodSuffix[j] == m) {
                    goodSuffix[j] = m - 1 - i;
                }
            }
        }
    }
    for (int i = 0; i < m - 1; i++) {
        goodSuffix[m - 1 - suffixes[i]] = m - 1 - i;
    }
    return goodSuffix;
}

private static int[] computeSuffixes(String pattern) {
    int m = pattern.length();
    int[] suffixes = new int[m];
    suffixes[m - 1] = m;
    int g = m - 1;
    int f = 0;
    for (int i = m - 2; i >= 0; --i) {
        if (i > g && suffixes[i + m - 1 - f] < i - g) {
            suffixes[i] = suffixes[i + m - 1 - f];
        } else {
            if (i < g) {
                g = i;
            }
            f = i;
            while (g >= 0 && pattern.charAt(g) == pattern.charAt(g + m - 1 - f)) {
                --g;
            }
            suffixes[i] = f - g;
        }
    }
    return suffixes;
}

// Aho-Corasick Algorithm
public static List<Integer> ahoCorasickSearch(String text, String[] patterns) {
    List<Integer> matches = new ArrayList<>();
    AhoCorasickNode root = buildAhoCorasickTrie(patterns);
    AhoCorasickNode state = root;

    for (int i = 0; i < text.length(); i++) {
        while (state != root && !state.children.containsKey(text.charAt(i)))
            state = state.fail;

        if (state.children.containsKey(text.charAt(i))) {
            state = state.children.get(text.charAt(i));
            if (state.outputs != null) {
                for (String found : state.outputs) {
                    matches.add(i - found.length() + 1);
                }
```

```java
       }
     }
   }
   return matches;
 }

 private static AhoCorasickNode buildAhoCorasickTrie(String[] patterns) {
   AhoCorasickNode root = new AhoCorasickNode();
   for (String pattern : patterns) {
     AhoCorasickNode node = root;
     for (char c : pattern.toCharArray()) {
       node.children.putIfAbsent(c, new AhoCorasickNode());
       node = node.children.get(c);
     }
     if (node.outputs == null) {
       node.outputs = new HashSet<>();
     }
     node.outputs.add(pattern);
   }

   Queue<AhoCorasickNode> queue = new LinkedList<>();
   for (Map.Entry<Character, AhoCorasickNode> entry : root.children.entrySet()) {
     entry.getValue().fail = root;
     queue.add(entry.getValue());
   }

   while (!queue.isEmpty()) {
     AhoCorasickNode current = queue.remove();
     for (Map.Entry<Character, AhoCorasickNode> entry : current.children.entrySet())
{
       AhoCorasickNode child = entry.getValue();
       AhoCorasickNode fail = current.fail;
       while (fail != null && !fail.children.containsKey(entry.getKey())) {
         fail = fail.fail;
       }
       if (fail == null) {
         child.fail = root;
       } else {
         child.fail = fail.children.get(entry.getKey());
         if (child.fail.outputs != null) {
           if (child.outputs == null) {
             child.outputs = new HashSet<>();
           }
           child.outputs.addAll(child.fail.outputs);
         }
       }
       queue.add(child);
     }
   }
   return root;
 }

 static class AhoCorasickNode implements Comparable<AhoCorasickNode> {
   TreeMap<Character, AhoCorasickNode> children = new TreeMap<>();
   AhoCorasickNode fail;
   Set<String> outputs;

   AhoCorasickNode() {
     this.fail = null;
     this.outputs = new HashSet<>();
   }
```

```java
    @Override
    public int compareTo(AhoCorasickNode other) {
      return this.hashCode() - other.hashCode();
    }
  }

  // Stream-based DNA sequence generator
  static class DNASequenceStream {
    private final long length;
    private long position;

    DNASequenceStream(long length) {
      this.length = length;
      this.position = 0;
    }

    public String nextChunk() {
      if (position >= length) return null;
      int chunkSize = (int) Math.min(CHUNK_SIZE, length - position);
      StringBuilder chunk = new StringBuilder(chunkSize);
      for (int i = 0; i < chunkSize; i++) {
        chunk.append(DNA_BASES.charAt(random.nextInt(DNA_BASES.length())));
      }
      position += chunkSize;
      return chunk.toString();
    }

    public void reset() {
      position = 0;
    }
  }

  // Helper Method to generate random DNA patterns
  private static String[] generateRandomPatterns(int length, int count) {
    String[] patterns = new String[count];
    for (int i = 0; i < count; i++) {
      StringBuilder pattern = new StringBuilder(length);
      for (int j = 0; j < length; j++) {
        pattern.append(DNA_BASES.charAt(random.nextInt(DNA_BASES.length())));
      }
      patterns[i] = pattern.toString();
    }
    return patterns;
  }

  private static long searchPatternStreamKMP(DNASequenceStream dnaStream, String
pattern) {
    long totalMatches = 0;
    String chunk;
    dnaStream.reset();

    while ((chunk = dnaStream.nextChunk()) != null) {
      List<Integer> matches = kmpSearch(chunk, pattern);
      totalMatches += matches.size();
    }

    return totalMatches;
  }

  private static long searchPatternStreamBM(DNASequenceStream dnaStream, String
pattern) {
    long totalMatches = 0;
```

```java
   String chunk;
   dnaStream.reset();

   while ((chunk = dnaStream.nextChunk()) != null) {
     List<Integer> matches = boyerMooreSearch(chunk, pattern);
     totalMatches += matches.size();
   }

   return totalMatches;
 }

 private static long searchPatternsStreamAC(DNASequenceStream dnaStream, String[]
patterns) {
   long totalMatches = 0;
   String chunk;
   dnaStream.reset();
   AhoCorasickNode root = buildAhoCorasickTrie(patterns);

   while ((chunk = dnaStream.nextChunk()) != null) {
     List<Integer> matches = ahoCorasickSearch(chunk, patterns);
     totalMatches += matches.size();
   }

   return totalMatches;
 }

 private static void printTrie(AhoCorasickNode node, String prefix, String
childPrefix) {
   System.out.println(prefix + (node.outputs != null ? node.outputs : "[]"));
   for (Map.Entry<Character, AhoCorasickNode> entry : node.children.entrySet()) {
     boolean isLast = entry.getKey().equals(node.children.lastKey());
     System.out.println(childPrefix + (isLast ? "└── " : "├── ") + entry.getKey());
     printTrie(entry.getValue(), childPrefix + (isLast ? "    " : "│   "), childPrefix
+ (isLast ? "    " : "│   "));
   }
 }

 private static int countTrieNodes(AhoCorasickNode node) {
   int count = 1;
   for (AhoCorasickNode child : node.children.values()) {
     count += countTrieNodes(child);
   }
   return count;
 }

}
```

CONCLUSION:

This project compared several string-matching algorithms for DNA sequence analysis, including EPMSPP, Ukkonen's Suffix Tree, Rabin-Karp, Z-Algorithm, KMP, Boyer-Moore, and Aho-Corasick. Our findings revealed that DNA-specific algorithms like EPMSPP offer significant advantages in genomic data processing, while general-purpose algorithms like Rabin-Karp show excellent scalability for large datasets. Algorithms with heavy preprocessing, such as Ukkonen's Suffix Tree, demonstrated a trade-off between initial construction time and subsequent query speed. For multi-pattern matching, Aho-Corasick proved superior, highlighting its importance in complex bioinformatics applications.

The project's limitations include the use of synthetic DNA sequences, which may not fully represent real genomic data complexities. Memory constraints for large datasets, particularly with suffix tree-based approaches, and the lack of parallelization in our implementations are some weaknesses. Our focus on exact matching and the standard DNA alphabet limits the direct applicability to some real-world scenarios that require approximate matching or protein sequence analysis.

Future research should explore parallelization of these algorithms to enhance performance on multi-core systems. Developing hybrid algorithms that combine strengths of different approaches, such as integrating DNA-specific optimizations with efficient multi-pattern matching, could yield powerful new tools. Extending the algorithms to handle approximate matching and compressed data would increase their utility in practical bioinformatics applications. Finally, comprehensive benchmarking with real genomic datasets from various species would provide more realistic performance metrics and insights.

These findings and future directions lay a foundation for more efficient genomic data analysis tools, potentially accelerating advancements in personalized medicine and genetic research. The project underscores the importance of tailoring algorithm selection to specific use cases and data characteristics in bioinformatics.

<u>REFERENCES:</u>

R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," in IBM Journal of Research and Development, vol. 31, no. 2, pp. 249-260, March 1987, doi: 10.1147/rd.312.0249.

Bhukya, R., & Somayajulu, D. V. L. N. (2011). Exact multiple pattern matching algorithm using DNA sequence and pattern pair. *International Journal of Computer Applications*, 17(8), 32-38.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3), 249-260.

Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6), 333-340.

Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2), 249-260.

Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2), 323-350.

Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10), 762-772.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: Computer science and computational biology*. Cambridge University Press.