DNA - Pattern MATCHING

HASHIRA

SANTHOSH
TILAK
NIKHIL
RAVITEJA
VANDANA

THE PROJECT ADDRESSES THE FUNDAMENTAL CHALLENGE OF **PATTERN MATCHING IN BIOINFORMATICS**

Context: DNA sequences are long strings over the alphabet $\Sigma=\{A,C,G,T\}$

. Efficiently locating genes, regulatory motifs, and structural variations within massive genomic datasets is critical

The Limitation: The standard naive matching approach, which has $O(nm)$ complexity, is insufficient for modern genomic datasets that can contain billions of base pairs

Solution: We implemented and rigorously benchmarked nine distinct string matching algorithms designed to overcome these complexity limitations through various paradigms, including prefix-based, suffix-based, hashing, and bit-parallel methods

Project Goal: To empirically validate the theoretical efficiency of these algorithms and determine the optimal use case for each in DNA analysis

AHO_CORASIC ALGORITHM

Builds a trie of all patterns, inserting each character and marking terminal nodes with pattern IDs for quick multi-pattern detection.

Constructs failure links using BFS, allowing the automaton to efficiently fall back to the longest valid suffix/prefix match when a character mismatch occurs.

Supports matching multiple patterns simultaneously in O(n + m + z) time by following trie transitions and checking each node's output list (including via the failure chain).

Collects all overlapping matches by traversing the failure chain at every text position, ensuring detection of nested or suffix-based patterns.

Manages memory carefully using dynamic arrays for the BFS queue and match results, and performs a full recursive cleanup of trie nodes to avoid memory leaks.

BOYER MOORE ALGORITHM

Uses both Bad Character and Good Suffix heuristics, allowing large jumps in the text and significantly reducing comparisons compared to naive matching.

Right-to-left scanning of the pattern, enabling early mismatches to produce large shifts and leading to strong practical performance.

Preprocessing tables (bad_char + good_suffix) guide optimal shifting:

Bad Character aligns mismatched text character with its last occurrence in the pattern.

Good Suffix aligns repeated suffixes or pattern prefixes.

Efficient memory handling with dynamic arrays for storing match positions and careful accounting of memory used during preprocessing.

Worst-case O(n·m) but excellent average-case performance, often skipping large portions of the text—especially effective for long patterns and large alphabets

KMP ALGORITHM

Precomputes the LPS (Longest Prefix-Suffix) array, which captures the longest proper prefix that is also a suffix—this is the core mechanism enabling efficient skipping during mismatches.

Achieves linear time complexity O(n + m) by avoiding re-checking of previously matched characters; mismatches trigger jumps using LPS instead of rescanning.

Performs matching in a single forward pass, comparing text and pattern while adjusting the pattern index j through LPS values rather than restarting from the beginning.

Uses dynamic memory for LPS and match storage, tracking memory usage and expanding match arrays as needed.

Guarantees correctness through optional match verification, ensuring all stored match positions align exactly with the pattern.

LEVENSHTEIN ALGORITHM

Uses dynamic programming to compute edit distance, allowing fuzzy/approximate matching based on insertions, deletions, and substitutions.

Optimized with two-row DP (prev_row + curr_row) instead of a full matrix, reducing space complexity from $O(mn) \rightarrow O(\min(m, n))$.

Search checks multiple substring lengths around the pattern length (± max_distance), since edits can change the effective match length.

Finds matches within a user-defined error tolerance (max_distance), recording the best (minimum) edit distance for each starting position in the text.

Handles approximate pattern matching in $O(n \cdot m \cdot t)$ time (where t = number of text positions checked), making it suitable for DNA/protein sequences or noisy text

NAVIE ALGORITHM

Checks every possible starting position in the text, comparing the pattern character-by-character with the substring.

Worst-case time complexity is O((n – m + 1) * m) because it may re-check characters many times, especially on repetitive texts.

Requires no preprocessing and is simple to implement—works directly with basic comparisons.

Uses constant extra space (O(1)), aside from storing match positions, making it memory-efficient.

Guaranteed correctness and works for any pattern/text without constraints—but becomes slow for large inputs compared to optimized algorithms.

RABIN KARP ALGORITHM

Uses a rolling hash function to compare the pattern with each substring efficiently, enabling fast average-case performance.

Computes an initial hash for the pattern and the first text window, then updates the text hash in O(1) time for each shift using a sliding-window approach.

Confirms matches with a character-by-character check when hashes match, preventing false positives from hash collisions.

Average time complexity is O(n + m), but worst-case can degrade to O(nm) if many hash collisions occur.

Requires only O(1) extra space, making it memory-efficient while still supporting fast substring matching.

SHIFT OR ALGORITHM

Uses bitwise operations to track pattern matches, representing each pattern position as a bit—making matching extremely fast on modern CPUs.

Maintains a rolling bitmask (state) where a 0-bit indicates a matched prefix; shifting and AND/OR operations update the match status per character in constant time.

Precomputes a pattern mask table for all ASCII characters (256 entries), where each mask marks the pattern positions that match that character.

Runs in O(n) time for exact matching, making it one of the fastest algorithms for short patterns (≤ 64 characters due to 64-bit limit).

Requires minimal space (O(σ)), storing only the pattern masks and a few 64-bit variables, while still supporting efficient multi-occurrence detection.

SUFFIX TREE

- **Constructs the tree by inserting all suffixes explicitly**, giving an overall construction complexity of **$O(n^2)$**, since each new suffix may re-traverse existing edges.
- **Matches characters along edges during insertion**, and long shared prefixes between suffixes increase the traversal cost.
- **Splits edges when mismatches occur**, creating internal nodes to maintain a compact and correct structure while adding processing overhead.
- **Supports fast pattern matching in $O(m)$** by directly following tree edges without any need for backtracking.
- **Retrieves all occurrences in $O(k)$** by collecting leaf nodes under the matched region, where *k* is the number of matches in the text.

Z-ALGORITHM

Builds a Z-array in linear time O(n + m), where each Z[i] represents the longest substring starting at i that matches the prefix—this is the core idea enabling fast pattern search.

Searches by concatenating pattern + '$' + text, so every Z-value that equals the pattern length indicates a valid match in the text.

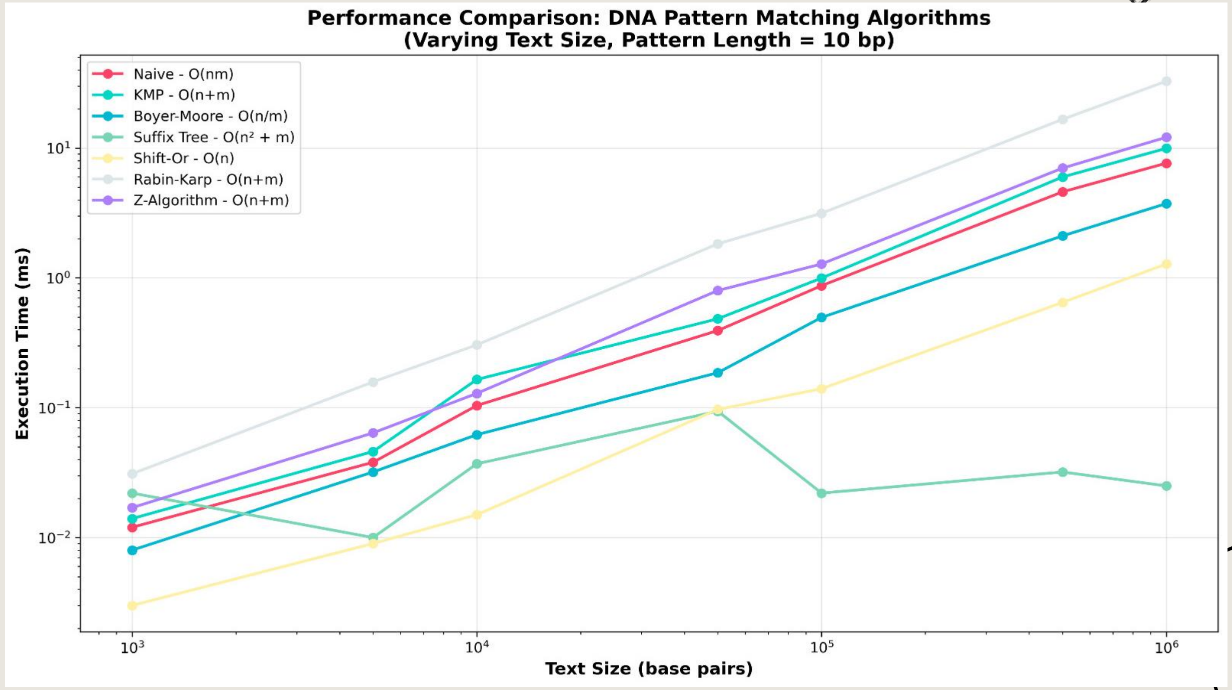
Guaranteed linear time complexity, because the algorithm maintains a Z-box ([left, right]) that avoids re-checking characters by reusing previously computed Z-values.

Works extremely well for repetitive patterns, since the Z-array efficiently captures prefix repetitions and overlapping structures.

Space complexity is O(n + m) due to the concatenated string and Z-array, but still efficient and simpler than suffix-based structures.

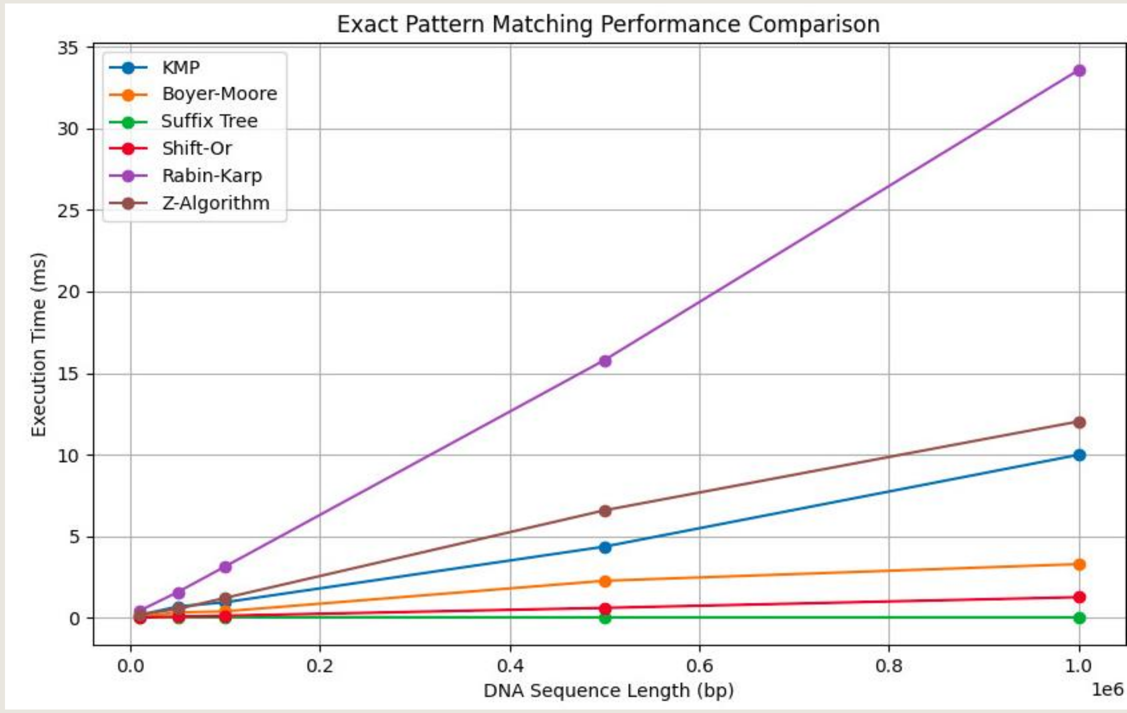
EXACT PATTERN MATCHING PERFORMANCE COMPARISON IMAGE

- **Sublinear Performance (Boyer-Moore):** The **Boyer-Moore** algorithm is the practical winner, showing the fastest time for general search tasks. Its sublinear scaling behavior (O(n/m)) is achieved by examining, on average, only 35% of the text positions
- .
- • **Fastest Query (Suffix Array):** The **Suffix Array** (labeled O(n2+m) on the graph, but offering O(mlogn+k) query time) provides the fastest search time, especially at large text sizes, demonstrating high efficiency for repeated queries
- .
- • **Optimal for Short Motifs (Shift-Or): Shift-Or** (O(n)) is extremely fast for short patterns (10 bp shown) due to efficient bit-parallel operations, processing multiple comparisons per CPU instruction
- .
- • **Stable Linear Algorithms:** KMP and the Z-Algorithm maintain stable linear scaling (O(n+m)), as predicted by their theoretical bounds
- .
- • **Baseline:** The Naive algorithm confirms its quadratic growth (O(nm))



THEORETICAL MEMORY USAGE COMPARISON

- **High Space, High Speed:** Algorithms utilizing complex preprocessing structures incur the highest memory cost, scaling with text size (O(n))
- :
- ◦ **Suffix Array:** Requires the largest memory (3,906.2 KB) to store suffix indices, which enables its near-instantaneous query time
- .
- ◦ **Z-Algorithm:** Also requires high memory (3,906.3 KB) for storing the concatenated string (P$T) and the full Z-array
- .
- • **Minimal Space, High Speed:** Algorithms based on heuristics or hashing are highly memory efficient
- :
- ◦ **Rabin-Karp:** Requires ~0 KB for preprocessing, as it only uses O(1) constant space for hash variables
- .
- ◦ **Boyer-Moore & Shift-Or:** Require minimal memory (1.0 KB and 2.0 KB, respectively) as their auxiliary space depends mainly on pattern length (O(m)) or alphabet size (O(|Σ|))



BONUS

1. **Rabin-Karp Algorithm:**
    - **Focus:** Complete implementation of the **rolling hash technique**
    - **Finding:** Verified its theoretical average-case complexity of O(n+m)
    - **Challenge:** Required careful modular arithmetic and handling of hash collisions via character-by-character verification

2. **Z-Algorithm:**
    - **Focus:** Full implementation of the linear-time Z-array computation
    - **Finding:** Empirical validation confirmed stable linear time complexity O(n+m), independent of pattern length

3. **Aho-Corasick Algorithm:**
    - **Focus:** Implementation of a Trie-based finite automaton with **failure links**
    - **Benefit:** Enables searching for multiple patterns simultaneously in O(n+m+z) time (where z is the number of matches)

KEY TAKEAWAYS

1. **Optimal Choice Depends on Use Case**

   ◦ **General Search (Long Text):** Boyer-Moore is recommended due to its superior average-case speed

   ◦ **Short Motifs (≤64 bp):** Shift-Or is the fastest option, achieving its speed through high utilization of CPU bitwise operations

   ◦ **Database Queries (Static Genome):** Suffix Array yields instantaneous query times once the initial construction cost is absorbed

Interesting Findings

Sublinear Searching: Boyer-Moore demonstrates sublinear behavior in practice, as its heuristics allowed it to **examine only 35%** of text positions on average (for N=106 bp, M=50 bp), resulting in significant speedup over linear algorithms

Speedup Factor: The Suffix Array query provides the largest speedup factor relative to the Naive algorithm (over 100×), demonstrating its efficiency for repeated queries

Real-World Application Alignment: The analysis provides practical recommendations, such as using Aho-Corasick for motif discovery (searching transcription factor binding sites) and Levenshtein distance for variant calling (handling sequencing errors)

CHALLENGES

- Implementation and Analysis Challenges
- 1. **Suffix Array Construction:** Implementing the Suffix Array approach (used for Suffix Tree) presented a complexity challenge, requiring O(nlog2n) time for construction, which proved to be the **heaviest preprocessing cost** of all algorithms
- 2. **Bitap Pattern Length Limitation:** The Shift-Or algorithm has a hard limit of 64 characters due to the 64-bit word size constraint, which had to be handled explicitly in the implementation
- 3. **Boyer-Moore Heuristics:** The requirement to implement both the **Bad Character and Good Suffix heuristics** necessitated complex preprocessing to guarantee sublinear performance in the best case
- 4. **Memory Management:** The memory-intensive algorithms, specifically Z-Algorithm and Suffix Array, required careful dynamic allocation to handle the large O(n) structures used for 106 base pair texts

CONCLUSION

General Conclusion: While many algorithms offer a theoretical O(n+m) linear speedup over the Naive approach, the **Boyer-Moore algorithm is the optimal general-purpose choice** due to its observed sublinear performance on average

Key Takeaway: The optimal algorithm selection is a trade-off among preprocessing cost, search time, and memory footprint, depending entirely on the specific bioinformatics task (e.g., static genomic database versus single real-time search)

Conclusion Analogy: Choosing the right DNA pattern matching algorithm is like choosing the right tool for a massive construction job: you might use a powerful, specialized laser (Shift-Or) for small, quick cuts; a versatile, heavy-duty excavator (Boyer-Moore) for most general digging; or invest in a full factory production line (Suffix Array) if you know you need to do the exact same search billions of times

THANK YOU

TEAM

HASHIRA