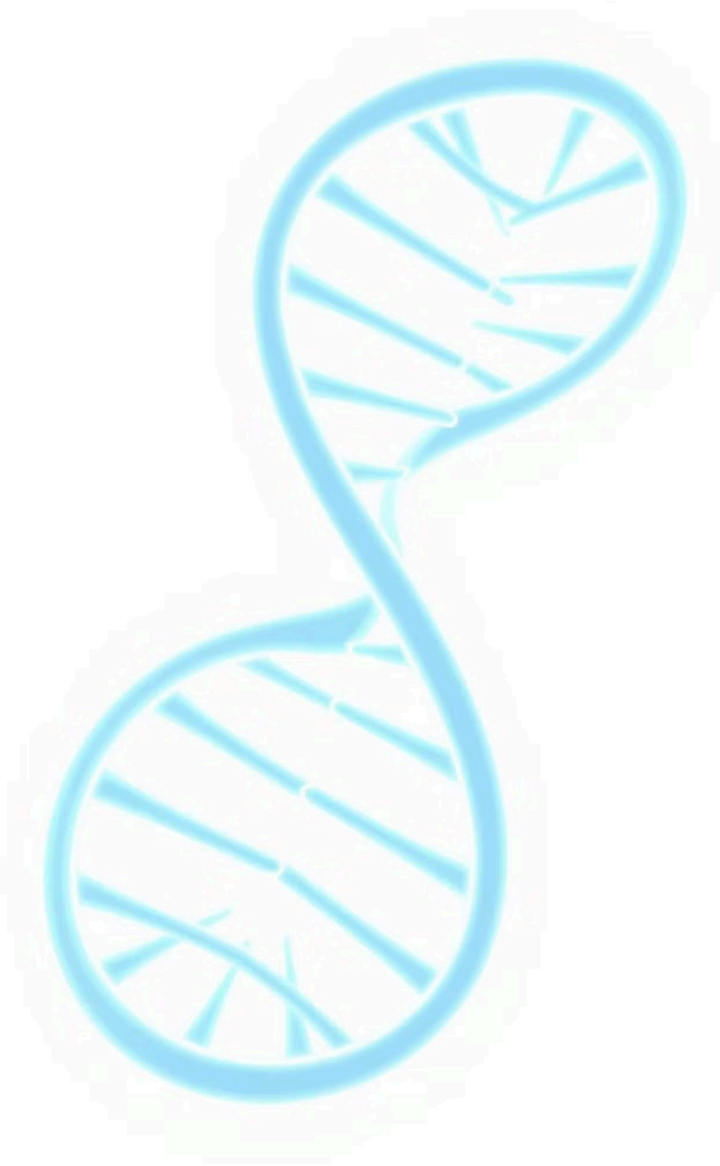
Project 7: Genome Sequence Storage, Query & Pattern Discovery Tool

Domain: Bioinformatics

Presented By: Yusra Haris & Fatima Mumtaz

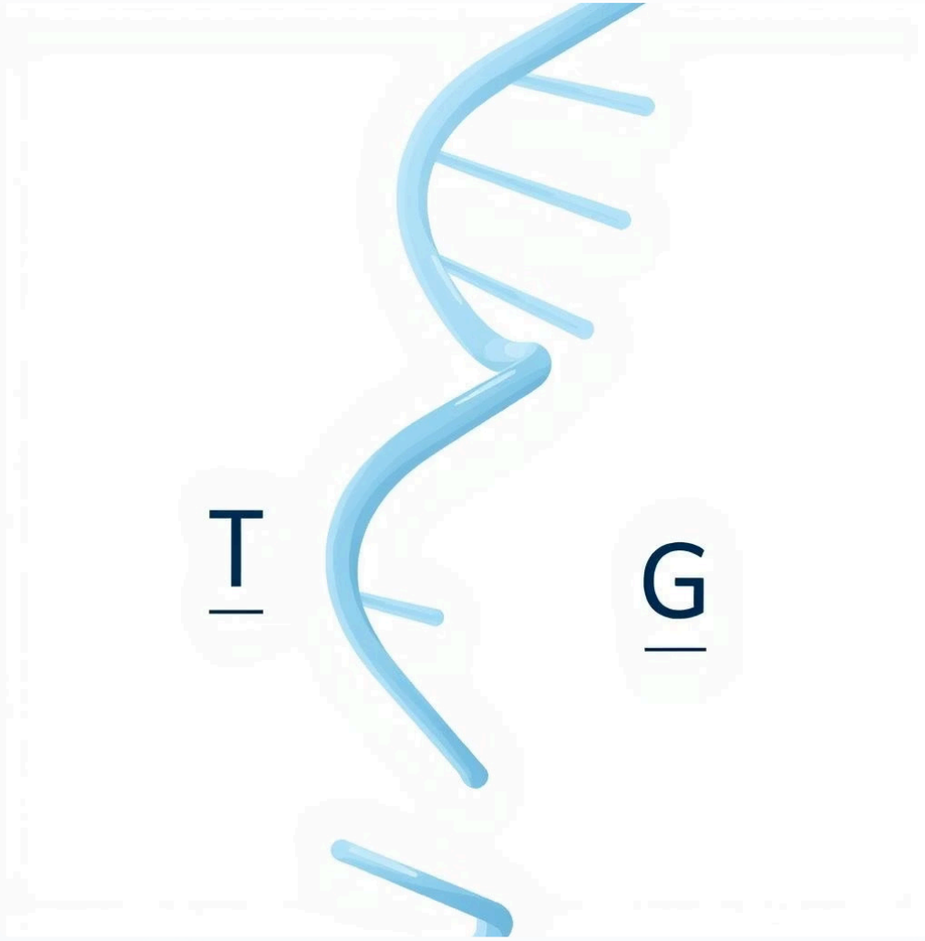


Domain Study: Understanding DNA Sequencing

DNA, the blueprint of life, is composed of four fundamental nucleotides: **Adenine (A), Thymine (T), Guanine (G), and Cytosine (C)**.

The human genome alone boasts approximately **3 billion base pairs**, a colossal amount of data that demands efficient methods for storage and rapid pattern search.

Analyzing these complex sequences unlocks critical insights across various fields:



Medical Applications

- Disease diagnosis (e.g., cancer markers)
- Drug development & personalized medicine
- Pathogen identification

Research Applications

- Evolutionary & genetic ancestry studies
- Agricultural genetics
- Forensic analysis

The Core Challenge: Efficient Genomic Analysis

> The primary hurdle in bioinformatics is the ability to **search billions of letters for specific patterns in a reasonable timeframe**.

Our Goal: Building a Robust Sequence Analysis Tool

Sequence Storage

Efficiently store vast DNA sequences.

Pattern Matching

Perform fast exact & approximate searches.

Motif Discovery

Identify frequent genomic motifs.

Scalable Querying

Support complex and rapid queries.

Defining Functional Requirements

To achieve our goal, the system must meet these specific functional requirements:

01

FR1: Sequence Storage

Input: DNA sequences (strings of A, T, G, C)

Output: Efficient in-memory storage

Constraint: Support up to **1000 sequences**, each up to **1000 base pairs**.

02

FR2: Exact Pattern Matching

Input: Pattern string (e.g., "ATCG")

Output: All positions where the pattern occurs

Performance: Targeting **O(m) time** where m = pattern length.

03

FR3: Frequent Motif Discovery

Input: k (motif length), threshold (minimum frequency)

Output: List of k-mers appearing ≥ threshold times

Performance: Aiming for **O(n) time** where n = sequence length.

04

FR4: Approximate Matching

Input: Pattern, max edit distance (1-2)

Output: Similar patterns with small differences

Example: "ATCG" matches "ATGG" (1 substitution).

High-Level Architecture Overview

Our system design incorporates modular components for clarity and efficiency:

SequenceStore

Raw sequence storage using vectors.

Index Module

Pattern search using a trie data structure.

MotifFinder

Frequency counting via rolling hash.

QueryEngine

Coordinates all operations and interactions.

Data Flow Example

1

User Uploads DNA

Sequences are stored in SequenceStore.

2

Index Construction

System builds search index with Index Module (trie).

3

Pattern Query

User searches for "ATCG", QueryEngine queries Index.

4

Motif Discovery

User requests top k-mers, MotifFinder scans with rolling hash.

Strategic Data Structure Choices

Choosing the right data structure for each component is crucial for optimal performance:

SequenceStore	Vector/Dynamic Array	Fast random access, dynamic resizing.	O(1) access
Index	Trie (Prefix Tree)	Efficient prefix matching, shared prefixes.	O(m) search
MotifFinder	Hash Table	Fast frequency counting, O(1) lookup.	O(1) insert/find
Rolling Window	Queue/Deque	Efficient sliding window operations.	O(1) push/pop
Edit Distance	2D Array (DP)	Store subproblem results, avoid recomputation.	O(m×n) space

Design Principle: "Use the right tool for the job" — leveraging specialized data structures for optimal time and space efficiency in each functional area.

Approach Option 1: Trie-Based Index

Idea: Insert all sequence prefixes or substrings into a tree-like structure known as a Trie (or Prefix Tree). Each node represents a character, and paths from the root form sequences.

Advantages

- **Very fast exact lookups:** Achieves O(m) time complexity, where 'm' is the pattern length.
- **Prefix-based queries:** Naturally efficient for searching patterns that share common prefixes.
- **Extensibility:** Can be extended for approximate matching through Depth-First Search (DFS) with edit-distance pruning.

Disadvantages & Use Cases

Cons:

- High memory usage, especially for large alphabets (like DNA's 4 bases).
- Hard to scale efficiently for full suffix trie implementations due to size.

Best For:

- Exact pattern matching.
- Approximate search with a small maximum edit distance (k).

Approach Option 2: Rolling Hash–Based Index

Idea: Utilizes a sliding window and polynomial rolling hash function (Rabin-Karp style) to efficiently compare substrings.

Advantages

- **Very memory-efficient:** Does not store substrings directly, only their hashes.
- **Motif frequency counting:** Perfectly suited for identifying frequent k-mers.
- **Fast substring comparisons:** After initial hashing, comparisons are O(1).
- **O(1) hash updates:** For each new window, the hash can be updated in constant time.

Disadvantages & Use Cases

Cons:

- Requires robust collision handling (e.g., using multiple hash functions).
- Less suitable for complex approximate pattern matching.
- Requires careful implementation of modular arithmetic.

Best For:

- Frequent motif detection (k-mer counting).
- Fast substring existence checks in large datasets.

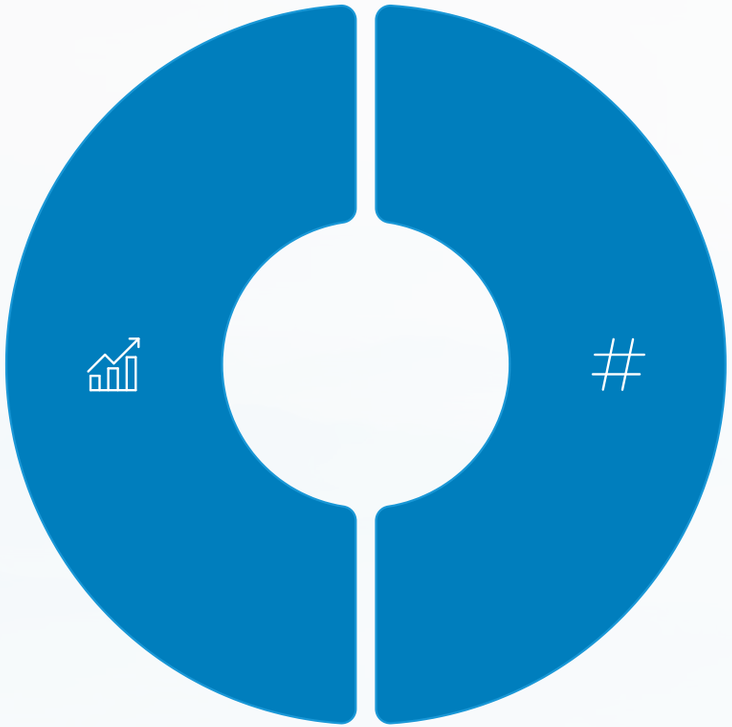
Final Decision: Hybrid Approach

Based on the analysis of both Trie and Rolling Hash methods, we recommend a hybrid strategy for optimal performance and functionality.

Trie Index

Selected for **exact & approximate substring search**.

- Fast lookups.
- Handles prefixes well.



Rolling Hash

Selected for **fast motif frequency detection**.

- Memory-efficient.
- Ideal for k-mer counting.

Why a Hybrid Approach?

- Provides the **best balance** of speed, memory usage, and implementation difficulty.
- Each module leverages its **optimal data structure** for specific tasks.
- Ensures the system **scales effectively** for the dataset sizes relevant to this project.

Project Plan & Risk Assessment

Our structured plan outlines key milestones and anticipates potential challenges.

Key Milestones

Week 9: Foundation

Project charter completion & sample sequence acquisition.

Week 10: Strategy

Presentation of the chosen hybrid approach.

Week 11: Implementation

Development of Trie index and Motif Finder modules.

Week 12: Validation

Demonstration of pattern query functionalities.

Week 13: Completion

Final integration, testing, and comprehensive report submission.

Anticipated Risks

→ High memory usage for Trie

Mitigation: Optimize node structure, consider memory-mapped files for larger datasets.

→ Hash collisions in Motif Module

Mitigation: Implement multiple hash functions or robust collision resolution strategies.

→ Increased complexity for approximate matching

Mitigation: Focus on well-bounded edit distances (1-2) to control time complexity.