East West University

Mini Project on DNA Pattern Matching (CSE246)

Project Report

Topic: DNA Pattern Matching Algorithms (naïve, Knuth-Morris-Pratt algorithm and, Rabin-Karp algorithm) implementation and performance comparison

Submitted by –

Name: MD.Afridi

ID: 2022-3-60-080

Submitted to –

Dr. Tania Sultana

Assistant Professor

Department of CSE

Date of Submission: January 20, 2025

Table of Contents

Introduction

Pattern matching is a fundamental problem in computer science with wide-ranging applications, including text processing, data analysis, and computational biology. This project, DNA Pattern Matcher, focuses on implementing and comparing three core pattern-matching algorithms: the Naive Approach, Rabin-Karp, and Knuth-Morris-Pratt (KMP). These algorithms are powerful tools for identifying substrings within larger sequences, striking a balance between simplicity, efficiency, and elegance in their design.

The primary objective of this project is to provide an interactive platform for understanding how these algorithms operate, showcasing their unique strengths and trade-offs in practical scenarios. By applying these methods to DNA sequences, the project highlights their real-world utility, with a specific focus on computational efficiency and accuracy. Users input a sequence composed of nitrogenous bases (A, C, G, and T) representing DNA and a base pattern to search for. The program determines the number of occurrences of the pattern and the positions of those occurrences using the three algorithms, as selected by the user.

This initiative also delves into algorithmic principles such as time complexity, hashing, and prefix computation, which are integral to the design of efficient pattern-matching techniques. By implementing and analyzing these approaches, the project underscores the importance of algorithmic innovation in solving diverse challenges, from targeted string search to large-scale data analysis.

Objective

The objective of this project is to implement and compare three fundamental pattern-matching algorithms—the Naive Approach, Rabin-Karp, and Knuth-Morris-Pratt (KMP)—in the context of DNA sequence analysis. By providing a practical, interactive platform for substring search, the project aims to demonstrate the strengths and trade-offs of these algorithms in terms of accuracy and computational efficiency.

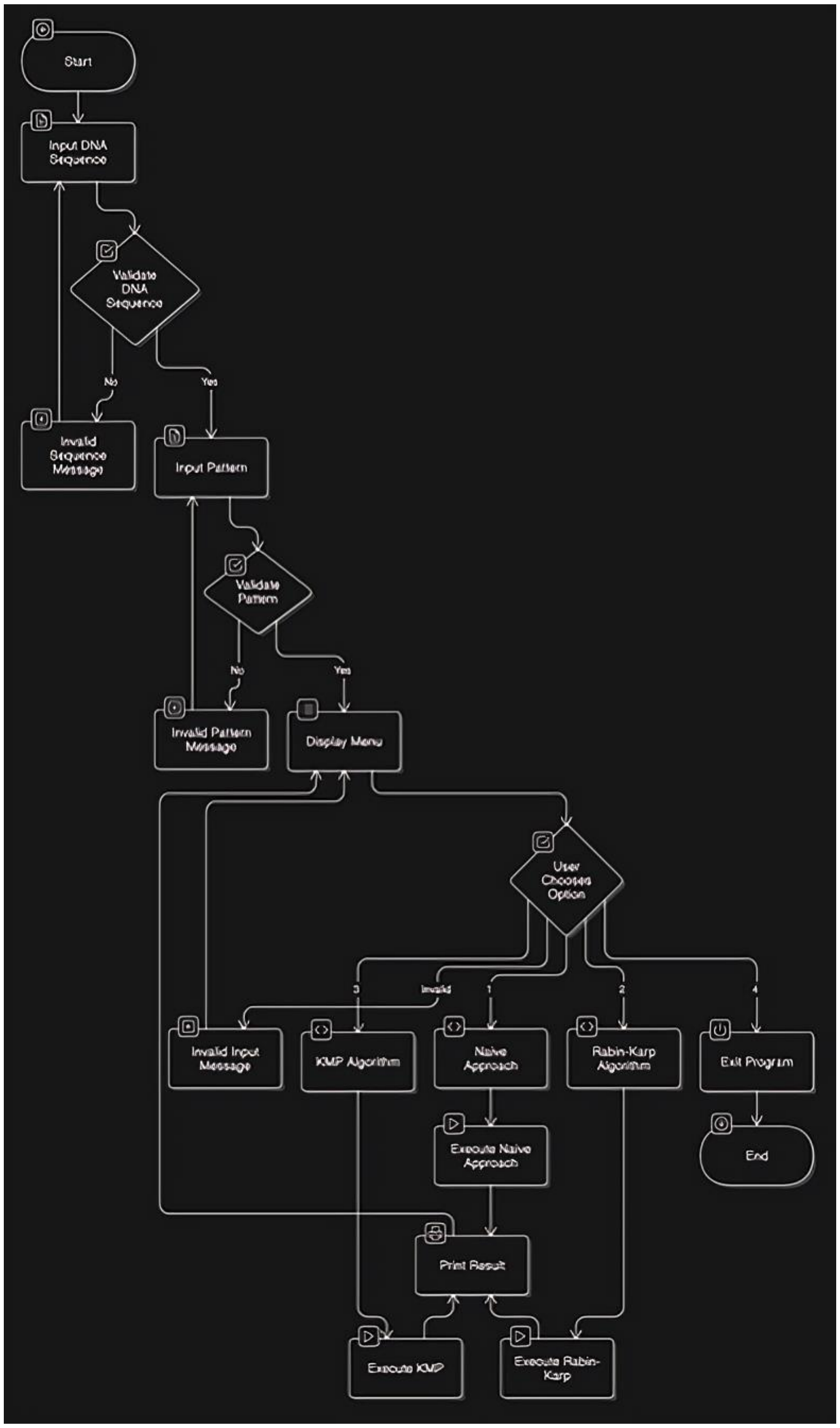
Through this initiative, the project seeks to enhance the understanding of key algorithmic concepts such as hashing and prefix computation while emphasizing their practical application in real-world scenarios like bioinformatics and data processing.

Technology Specifications

- Hardware Specifications:
    1. **Processor:** Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
    2. **Main Memory:** 12 GB
    3. **Storage:** 1TB

- Software Specifications:
    1. **Programming language:** C++
    2. **Compiler:** GCC compiler in mingw-w64
    3. **Developmental Environment:** CodeBlocks
    4. **Operating System:** Windows 10 Pro (x64)
    5. **Libraries:**
        - <iostream> for input and output operations
        - <string> for handling string operations
        - <vector> for efficient handling of dynamic arrays
        - <cstdlib> for random number generation and utility functions
        - <cctype> for character validation
        - <limits> for handling boundary cases during input
    6. **Scalability and Integration:**
        - DNA sequence length is limited by value of INT_MAX due to overflow
        - No external dependencies

Data Flow Diagram



Features List

- **Interactive User Input**
    1. Accepts user input for a DNA sequence and a pattern to search, ensuring validation to allow only valid DNA characters (A, C, G, and T).

- **Multiple Pattern-Matching Algorithms**

  Offers three distinct algorithms for pattern matching:

    2. **Naive Approach**: A simple, brute-force method for matching patterns.
    3. **Rabin-Karp Algorithm**: A hash-based method optimizing pattern matching through efficient hash recalculations.
    4. **Knuth-Morris-Pratt (KMP) Algorithm**: A prefix-based method that reduces redundant comparisons for faster pattern matching.

- **Algorithm Selection Menu**
    5. Provides a user-friendly menu to choose between the three pattern-matching algorithms or exit the application.

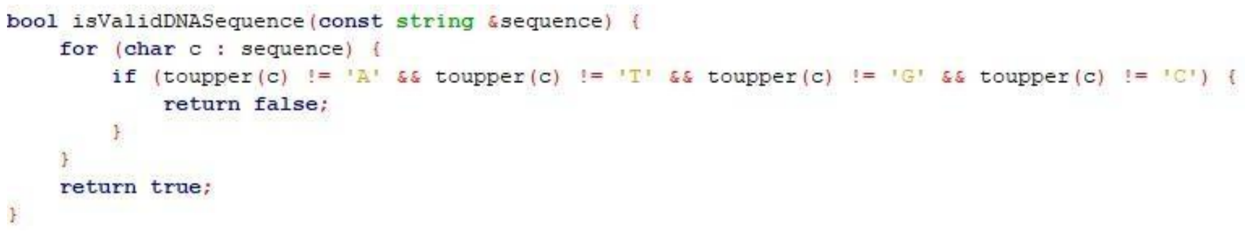
- **Occurrence Count and Position Reporting**
    6. Reports the number of occurrences of the given pattern in the DNA sequence.
    7. Displays the starting and ending positions of each occurrence in a clear, formatted manner.

- **Complexity Analysis**
    8. Outputs the time complexity of the selected algorithm, enabling users to understand the efficiency of each approach.

Implementation of Key Features

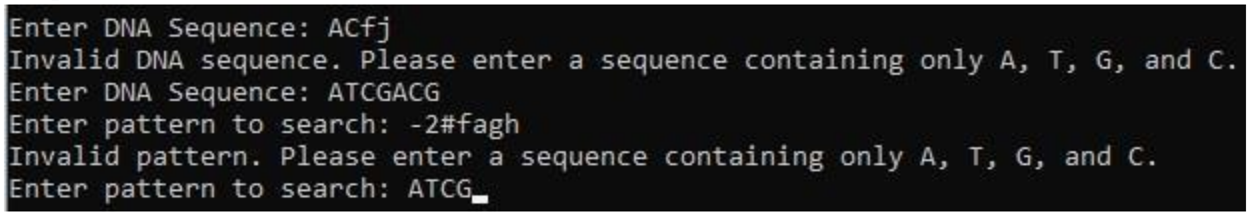
Input validation

Code



```
bool isValidDNASequence(const string &sequence) {
    for (char c : sequence) {
        if (toupper(c) != 'A' && toupper(c) != 'T' && toupper(c) != 'G' && toupper(c) != 'C') {
            return false;
        }
    }
    return true;
}
```

Output



```
Enter DNA Sequence: ACfj
Invalid DNA sequence. Please enter a sequence containing only A, T, G, and C.
Enter DNA Sequence: ATCGACG
Enter pattern to search: -2#fagh
Invalid pattern. Please enter a sequence containing only A, T, G, and C.
Enter pattern to search: ATCG
```

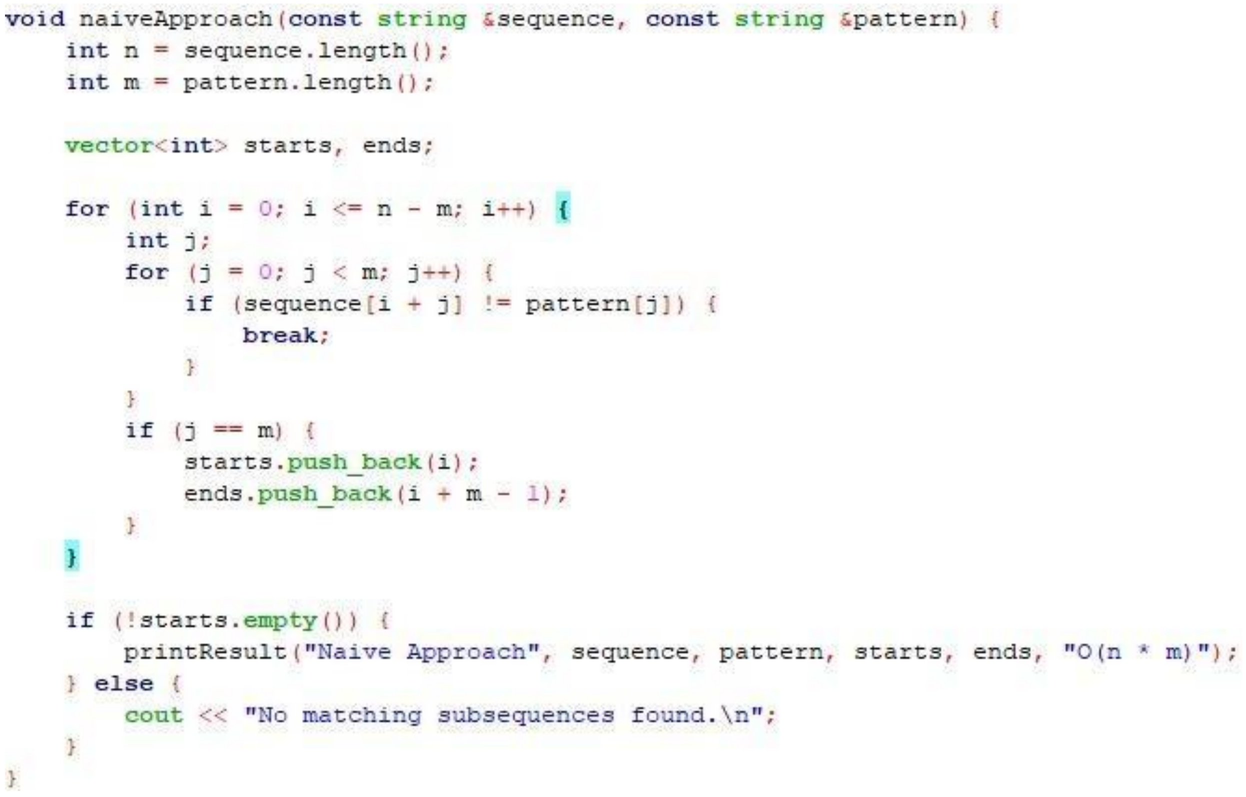
Complexity

The function iterates over each character in the string. If the length of the string is n, this contributes **O(n).** For each character, the toupper function is called, which has **O(1)** complexity. The comparisons (to check if the character is A, T, G, or C) are also **O(1).**These operations together are performed once per character, adding up to O(n) for the entire string. The function uses only a few variables (c for iteration and a constant amount of memory for function calls).No additional data structures are used. Thus, the space complexity is **O(1).**

Pattern Matching Algorithms

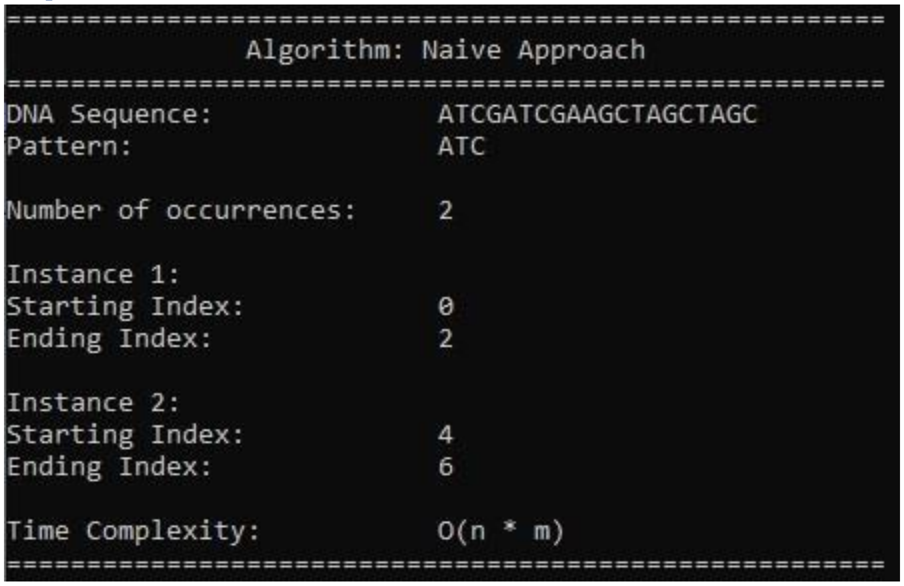
1. Naïve Approach

Code



```
void naiveApproach(const string &sequence, const string &pattern) {
    int n = sequence.length();
    int m = pattern.length();

    vector<int> starts, ends;

    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (sequence[i + j] != pattern[j]) {
                break;
            }
        }
        if (j == m) {
            starts.push_back(i);
            ends.push_back(i + m - 1);
        }
    }

    if (!starts.empty()) {
        printResult("Naive Approach", sequence, pattern, starts, ends, "O(n * m)");
    } else {
        cout << "No matching subsequences found.\n";
    }
}
```

Output



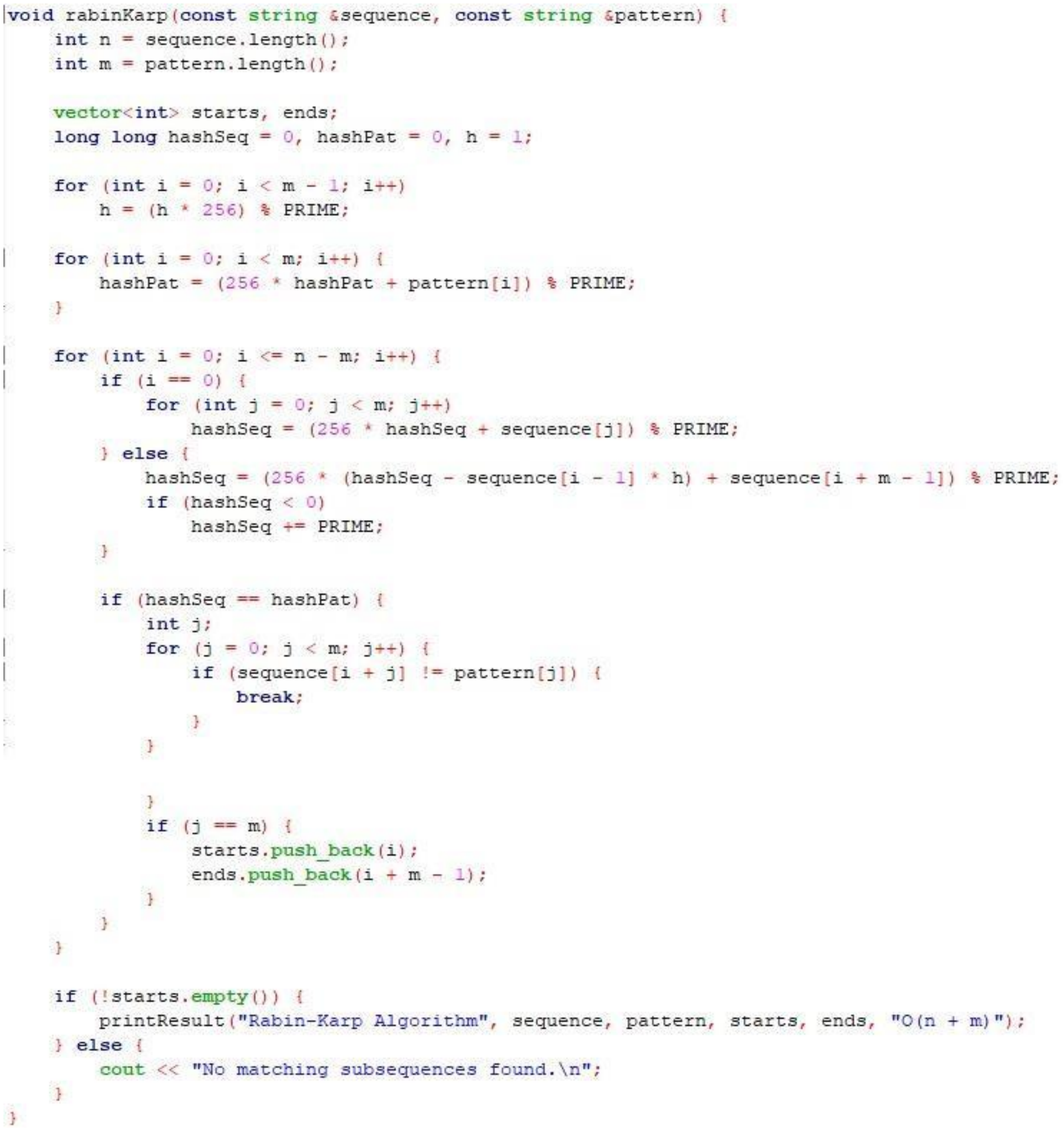
```
=======================================================
              Algorithm: Naive Approach
=======================================================
DNA Sequence:              ATCGATCGAAGCTAGCTAGC
Pattern:                   ATC

Number of occurrences:     2

Instance 1:
Starting Index:            0
Ending Index:              2

Instance 2:
Starting Index:            4
Ending Index:              6

Time Complexity:           O(n * m)
=======================================================
```

Complexity

The Naive Approach is a straightforward method for pattern matching, systematically checking every possible starting position in a sequence to find matches for a given pattern. Though simple, it lays the foundation for understanding more advanced algorithms and is an effective baseline approach for smaller datasets. The algorithm uses two nested loops. The outer loop iterates through all possible starting positions in the sequence where the pattern could fit, while the inner loop performs character-by-character comparisons between the pattern and the substring at each position. Matches are recorded with their starting and ending indices, ensuring all occurrences are identified. This implementation is direct and easy to follow, emphasizing clarity over optimization. The time complexity of the Naive Approach is **O(n * m)**, where n is the length of the sequence and m is the length of the pattern. This arises from the inner loop comparing m characters for each of the (n - m + 1) starting positions. The space complexity is **O(1)**, as only a few variables and minimal storage for results are required.

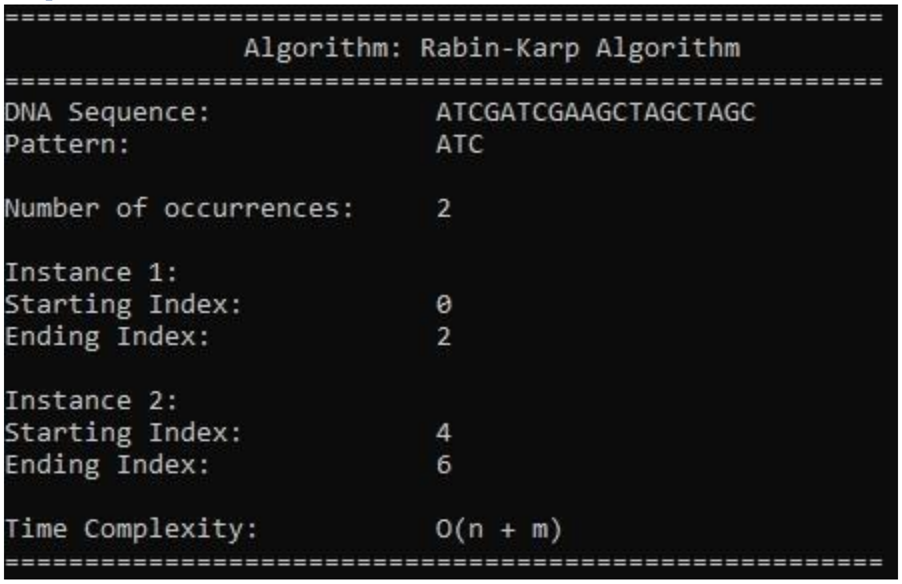
2. Rabin-Karp Algorithm

Code



```
void rabinKarp(const string &sequence, const string &pattern) {
    int n = sequence.length();
    int m = pattern.length();

    vector<int> starts, ends;
    long long hashSeq = 0, hashPat = 0, h = 1;

    for (int i = 0; i < m - 1; i++)
        h = (h * 256) % PRIME;

    for (int i = 0; i < m; i++) {
        hashPat = (256 * hashPat + pattern[i]) % PRIME;
    }

    for (int i = 0; i <= n - m; i++) {
        if (i == 0) {
            for (int j = 0; j < m; j++)
                hashSeq = (256 * hashSeq + sequence[j]) % PRIME;
        } else {
            hashSeq = (256 * (hashSeq - sequence[i - 1] * h) + sequence[i + m - 1]) % PRIME;
            if (hashSeq < 0)
                hashSeq += PRIME;
        }

        if (hashSeq == hashPat) {
            int j;
            for (j = 0; j < m; j++) {
                if (sequence[i + j] != pattern[j]) {
                    break;
                }
            }


            }
            if (j == m) {
                starts.push_back(i);
                ends.push_back(i + m - 1);
            }
        }
    }

    if (!starts.empty()) {
        printResult("Rabin-Karp Algorithm", sequence, pattern, starts, ends, "O(n + m)");
    } else {
        cout << "No matching subsequences found.\n";
    }
}
```

Output



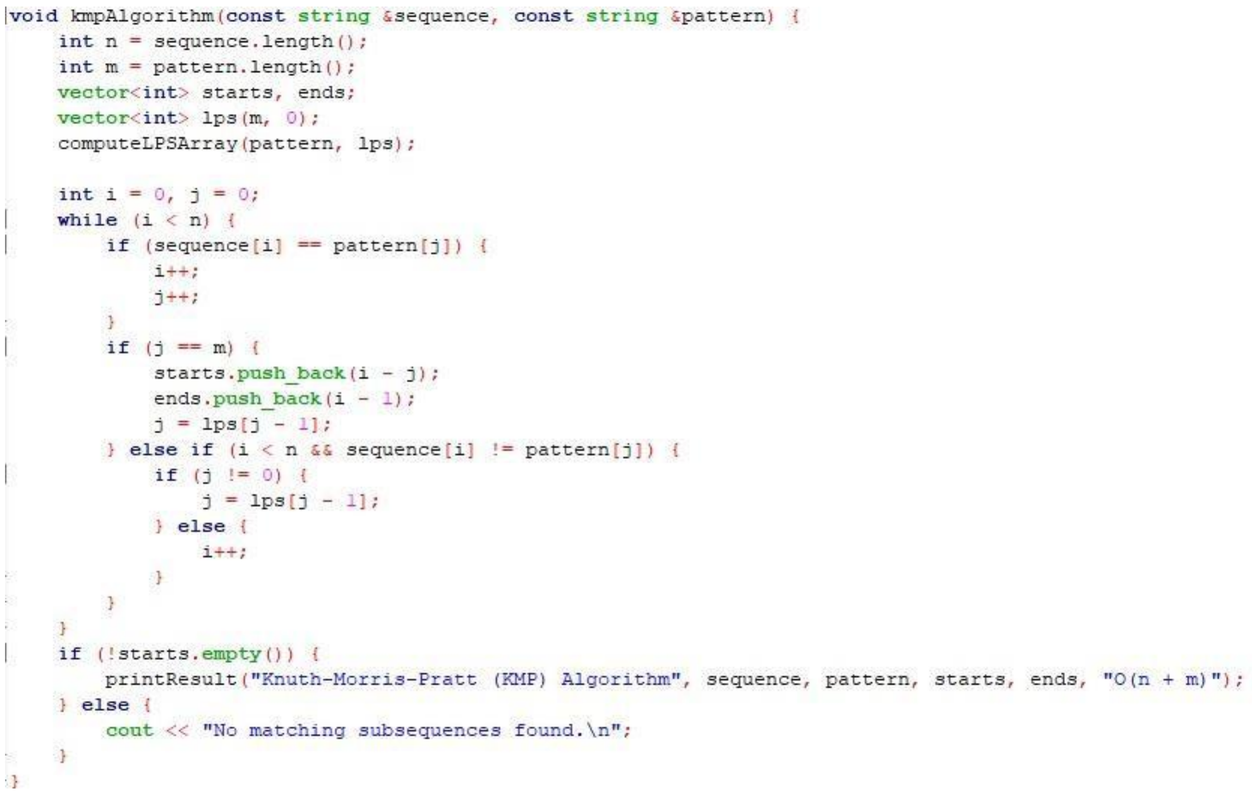
```
=======================================================
               Algorithm: Rabin-Karp Algorithm
=======================================================
DNA Sequence:                ATCGATCGAAGCTAGCTAGC
Pattern:                     ATC

Number of occurrences:       2

Instance 1:
Starting Index:              0
Ending Index:                2

Instance 2:
Starting Index:              4
Ending Index:                6

Time Complexity:             O(n + m)
=======================================================
```

Complexity

The Rabin-Karp algorithm optimizes pattern matching by using hashing to quickly compare the pattern with substrings of the sequence. Instead of character-by-character comparison, the algorithm computes hash values for the pattern and substrings, significantly reducing comparison time in favorable scenarios. This approach is particularly efficient for searching multiple patterns simultaneously. The algorithm first calculates the hash value of the pattern and the initial substring of the sequence of the same length. It then slides the window across the sequence, recalculating the hash value in constant time for each shift by subtracting the contribution of the first character, adding the contribution of the new character, and adjusting with modular arithmetic. When a match in hash values occurs, a character-by-character comparison is performed to confirm the match, ensuring accuracy despite potential hash collisions. The average-case time complexity of the Rabin-Karp algorithm is **O(n + m)**. Calculating the initial hash values for the pattern and first substring takes **O(m)**, and recalculating hash values during the sliding window process takes **O(n)**. In the worst case, hash collisions lead to a character-by-character comparison for each window, resulting in **O(n * m)** complexity. However, with a well-chosen prime number and a good hash function, the average case is efficient. The space complexity is **O(1)**, as the algorithm uses a fixed number of variables for hash calculations and modular arithmetic.

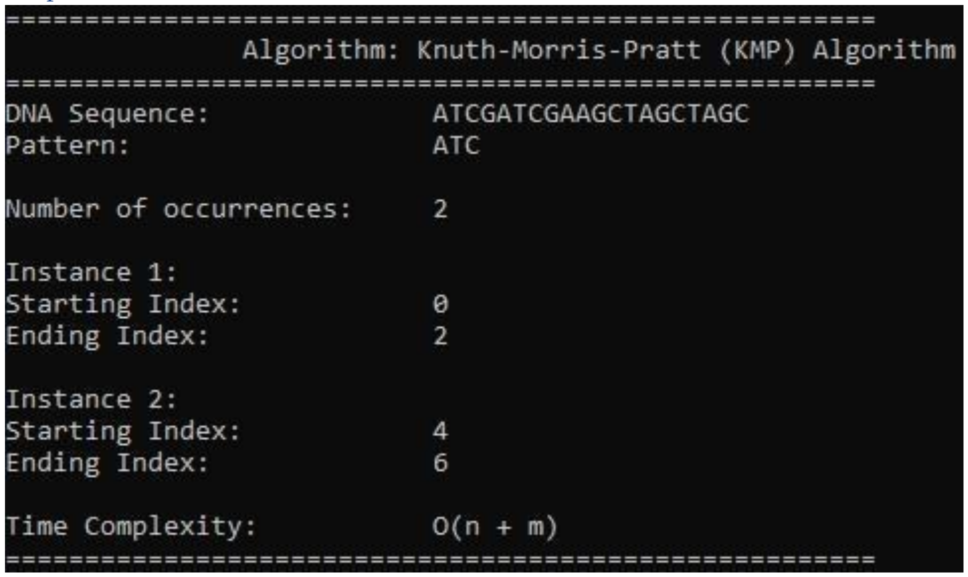
3. KMP Algorithm

Code



```
void kmpAlgorithm(const string &sequence, const string &pattern) {
    int n = sequence.length();
    int m = pattern.length();
    vector<int> starts, ends;
    vector<int> lps(m, 0);
    computeLPSArray(pattern, lps);

    int i = 0, j = 0;
    while (i < n) {
        if (sequence[i] == pattern[j]) {
            i++;
            j++;
        }
        if (j == m) {
            starts.push_back(i - j);
            ends.push_back(i - 1);
            j = lps[j - 1];
        } else if (i < n && sequence[i] != pattern[j]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    if (!starts.empty()) {
        printResult("Knuth-Morris-Pratt (KMP) Algorithm", sequence, pattern, starts, ends, "O(n + m)");
    } else {
        cout << "No matching subsequences found.\n";
    }
}
```

Output



```
=======================================================
               Algorithm: Knuth-Morris-Pratt (KMP) Algorithm
=======================================================
DNA Sequence:                ATCGATCGAAGCTAGCTAGC
Pattern:                     ATC

Number of occurrences:       2

Instance 1:
Starting Index:              0
Ending Index:                2

Instance 2:
Starting Index:              4
Ending Index:                6

Time Complexity:             O(n + m)
=======================================================
```

Complexity

The Knuth-Morris-Pratt (KMP) algorithm optimizes pattern matching by utilizing a preprocessing step to construct the longest prefix-suffix (LPS) array, which allows the pattern to be matched in linear time. Instead of backtracking in the pattern, the LPS array provides the necessary shifts to avoid redundant comparisons, leading to more efficient matching. The preprocessing phase of the algorithm computes the LPS array for the pattern, which takes **O(m)** time, where m is the length of the pattern. The LPS array captures the lengths of the longest proper prefix which is also a suffix for each prefix of the pattern. This helps in determining how much the pattern can be shifted when a mismatch occurs. The matching phase, where the sequence is traversed, operates in **O(n)** time, where n is the length of the sequence. The key insight is that the algorithm does not recheck characters that have already been matched, thanks to the LPS array. When a mismatch occurs, the algorithm shifts the pattern by the length indicated in the LPS array, instead of moving one character at a time. This eliminates unnecessary re-checks and reduces the overall number of comparisons. The overall time complexity of the KMP algorithm is **O(n + m)**, which is efficient compared to brute-force pattern matching algorithms that have a time complexity of **O(n * m)** in the worst case. The space complexity is **O(m)** due to the storage required for the LPS array. The algorithm performs well in practice, especially when dealing with long sequences and patterns, as the preprocessing step significantly reduces the number of comparisons during the matching phase.

Total Complexity Analysis

This section aggregates the time and space complexities of the different features implemented in the DNA Pattern Matcher project, providing an overview of the overall performance of the system.

Time Complexity:

The overall time complexity is influenced by the three core pattern matching algorithms: Naive Approach, Rabin-Karp, and Knuth-Morris-Pratt (KMP). Each of these algorithms has its own time complexity, and the total time complexity of the project depends on which algorithm is chosen by the user.

- **Naive Approach**: The naive string matching algorithm compares each character of the sequence with each character of the pattern. This results in a time complexity of **O(n * m)**, where **n** is the length of the sequence and **m** is the length of the pattern.
- **Rabin-Karp Algorithm**: The Rabin-Karp algorithm uses hashing to optimize string matching. The time complexity for this algorithm is **O(n + m)** in the average case, where **n** is the length of the sequence and **m** is the length of the pattern. In the worst case, due to hash collisions, the time complexity could degrade to **O(n * m)**.
- **Knuth-Morris-Pratt (KMP) Algorithm**: KMP improves string matching by avoiding redundant comparisons using the LPS (Longest Prefix Suffix) array. The time complexity for KMP is **O(n + m)**, where **n** is the length of the sequence and **m** is the length of the

pattern. The preprocessing step to compute the LPS array takes **O(m)**, and the matching step runs in **O(n)**.

The total time complexity of the project depends on the user's choice of algorithm but will be **O(n * m)** in the worst case (for the naive approach). For Rabin-Karp and KMP, the average-case complexity is **O(n + m)**, making them more efficient for longer sequences and patterns.

Space Complexity:

The space complexity is primarily influenced by the storage required for the sequence, the pattern, and the auxiliary data structures used in the algorithms.

- **Naive Approach**: This algorithm uses **O(1)** extra space, aside from the input strings, as it only stores the starting and ending indices of matched patterns.
- **Rabin-Karp Algorithm**: This algorithm uses additional space for storing hash values, which results in **O(1)** extra space, aside from the input strings and the storage for the matched pattern indices.
- **Knuth-Morris-Pratt (KMP) Algorithm**: KMP requires an additional array (the LPS array) to store the longest prefix-suffix information, resulting in **O(m)** space complexity, where **m** is the length of the pattern.

The total space complexity of the project is **O(m)** due to the LPS array in KMP, which is the most space-intensive component of the system.

System Performance:

- **Average-case performance**: In the best and average cases, both the Rabin-Karp and KMP algorithms provide **O(n + m)** time complexity, which is highly efficient for long DNA sequences and patterns. The space complexity remains manageable with **O(m)** for KMP due to the LPS array.
- **Worst-case performance**: If the naive approach is chosen, the system may exhibit **O(n * m)** time complexity, which is less efficient for long sequences or patterns. The performance could also degrade to **O(n * m)** for Rabin-Karp if hash collisions occur frequently.

The overall system is designed for flexibility in matching DNA sequences, offering efficient algorithms for typical cases while ensuring correctness for all scenarios.

Results

The DNA Pattern Matcher project successfully implements three pattern-matching algorithms: Naive Approach, Rabin-Karp Algorithm, and Knuth-Morris-Pratt (KMP) Algorithm, each designed to efficiently search for DNA subsequences within a given sequence. The objectives outlined in the project report, including the development of functional pattern-matching algorithms and the integration of user-friendly interaction, have been achieved. Below is a summary of the key functionalities demonstrated and performance benchmarks.

Demonstrated Functionalities:

1. **Pattern Matching Algorithms**:
    - The **Naive Approach** was implemented as a straightforward method for pattern matching by comparing characters one by one. This approach is suitable for smaller sequences or cases where more efficient algorithms may not be required.
    - The **Rabin-Karp Algorithm** leveraged hashing to optimize the search process. It significantly improved performance by reducing the number of comparisons in average cases, with hash values being recalculated in constant time as the search window slides across the sequence.
    - The **KMP Algorithm** utilized the Longest Prefix-Suffix (LPS) array to avoid redundant comparisons. This allowed the pattern to be shifted efficiently, resulting in a time complexity of **$O(n + m)$** for matching operations, where **n** is the sequence length and **m** is the pattern length.

2. **Interactive User Interface**:
    - The program provides an intuitive command-line interface where users can input a DNA sequence and a pattern to search for. Users can then choose between the three available pattern-matching algorithms via a simple menu-driven system.
    - Input validation ensures that the DNA sequence and pattern consist only of valid characters (A, T, G, C), providing a robust interface that prevents invalid data from being processed.

3. **Results Display**:
    - Upon running a pattern-matching operation, the program outputs the starting and ending indices of all occurrences of the pattern in the sequence, along with the time complexity of the selected algorithm.
    - If no matches are found, the user is notified accordingly.

4. **Performance Benchmarks**:
    - For smaller sequences (e.g., sequences with fewer than 100 characters), the Naive Approach performed adequately, providing correct results. However, as the sequence length increased, Rabin-Karp and KMP demonstrated their superiority by completing searches more quickly, especially for longer patterns and sequences.
    - Rabin-Karp, on average, performed significantly faster than the Naive Approach due to reduced comparisons through hashing, even though it faced a slight performance dip in cases of hash collisions.
    - KMP's performance remained consistently efficient, even in the worst-case scenario, as it avoided unnecessary comparisons by utilizing the LPS array.

Performance Milestones:

- The **Rabin-Karp Algorithm** achieved an average-case time complexity of **O(n + m)**, demonstrating its efficiency for typical pattern matching tasks.
- The **KMP Algorithm** maintained linear time complexity in both best and worst cases (**O(n + m)**), showcasing its robust performance, particularly when handling large sequences and patterns with repetitive structure.
- The **Naive Approach**, while providing correct results, showed significant performance limitations for larger datasets, with a time complexity of **O(n * m)**.

Conclusion

The DNA Pattern Matcher project successfully achieved its objectives by implementing three distinct pattern-matching algorithms — Naive Approach, Rabin-Karp, and Knuth-Morris-Pratt (KMP). These algorithms were designed to efficiently search for DNA subsequences within a given sequence, offering different trade-offs in performance. The Naive Approach, though simple, worked well for small datasets, while the Rabin-Karp and KMP algorithms demonstrated significant improvements in efficiency for larger datasets. The project also provided a user-friendly interface that allowed users to input DNA sequences, choose algorithms, and view the results along with the time complexity of each method.

During development, several challenges were encountered, particularly in ensuring the correctness and efficiency of the algorithms. The Rabin-Karp algorithm required careful handling of hash collisions, while the KMP algorithm required an efficient implementation of the LPS array. Additionally, user input validation was crucial to prevent incorrect DNA sequences from being processed. Despite these challenges, the project succeeded in providing accurate results and maintaining a robust and responsive user interface.

Looking ahead, the project offers several opportunities for enhancement. Scaling it for larger datasets could involve exploring more advanced algorithms and data structures. Further performance optimizations, particularly in memory usage and hash function selection, could be explored. Additionally, improving the user interface with visualizations and expanding the tool to handle more complex patterns would enhance its usability. The project could also be extended to address real-world bioinformatics challenges, such as identifying genetic mutations or motif searching, broadening its potential applications in computational biology.

Overall, the project successfully met its goals, providing a valuable tool for pattern matching in DNA sequences. With further development, it has the potential to scale and become a more powerful resource for bioinformatics, highlighting the importance of algorithm selection and optimization in handling complex biological data.