**Experiment No. 9**

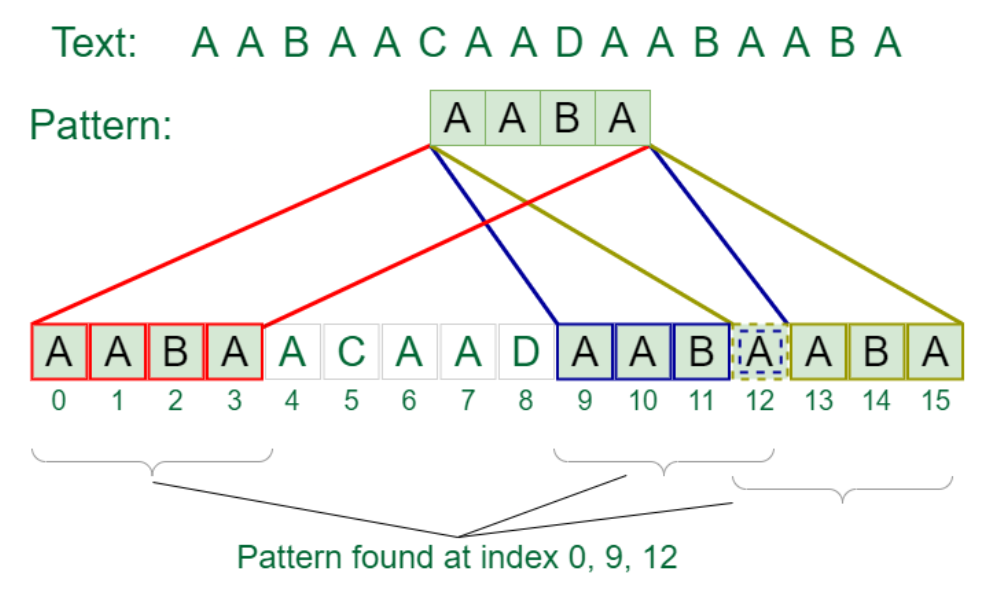
**Aim:** Implement following string-matching algorithms and analyze time complexities.

1. **Naïve Algorithm**
2. **Rabin Karp Algorithm**

**Objective:** To implement String Matching Algorithms.

**Theory:**

**Naïve Algorithm:** The Naive Algorithm, also known as the Brute-Force Algorithm, is a straightforward approach to pattern searching in a given text. The primary idea behind this algorithm is to check all possible positions of the text to find occurrences of a given pattern. It compares each character of the pattern with the corresponding character of the text and slides the pattern one position at a time until it finds a match or reaches the end of the text. If a match is found, it records the starting position of the match.



**Algorithm:**

1. Start from the beginning of the text and move the pattern one character at a time.
2. At each position of the text, compare each character of the pattern with the corresponding character in the text.
3. If all characters of the pattern match the corresponding characters in the text, a match is found.
4. Record the starting position of the match.
5. Continue the process until you reach the end of the text.
6. Slide the pattern one position to the right and repeat steps 2-5 until the end of the text is reached.
7. When the end of the text is reached, all positions with a complete match of the pattern are recorded.

**Time Complexity:**

The time complexity of the Naive Algorithm is **O((n-m+1)\*m)**, where:

* n is the length of the text.
* m is the length of the pattern.

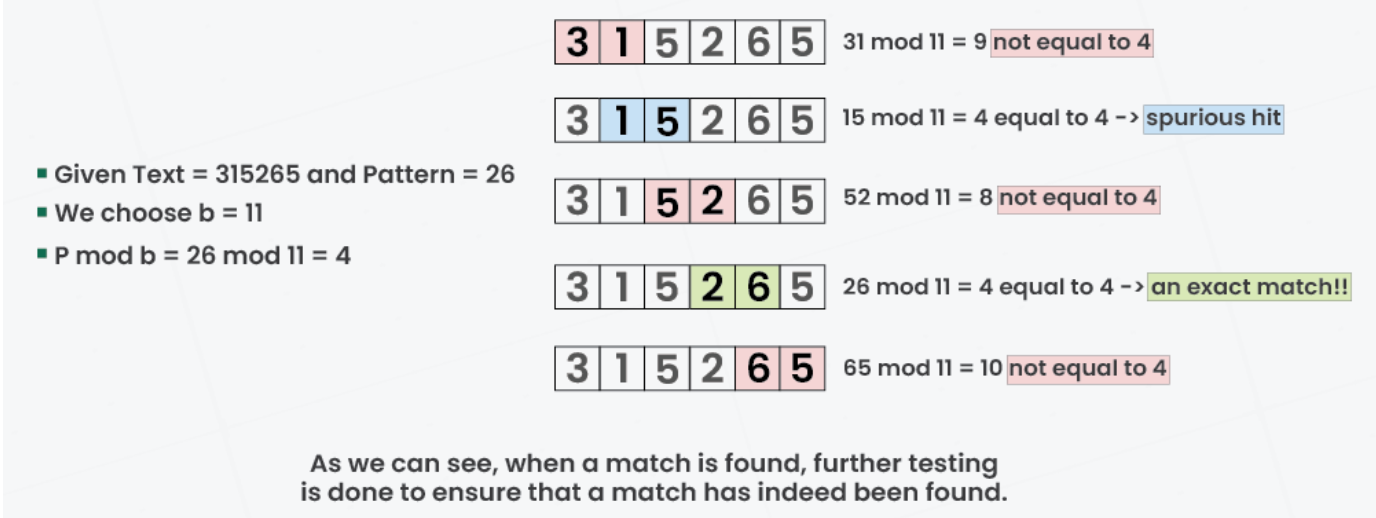
In the worst case, the algorithm needs to compare all characters in the pattern with all characters in the text at every position. This results in a nested loop structure, leading to the mentioned time complexity. The algorithm has a linear relationship with both the text and pattern lengths.

**Real World Applications:** The Naive Algorithm, despite its simplicity and relatively higher time complexity, has some practical applications:

1. **Text Editors and Word Processors:** The Naive Algorithm can be used for searching and replacing text in documents.
2. **Data Mining:** It can be used to search for specific patterns or sequences of data within large datasets.
3. **Search Engines:** Some search engines use variants of the Naive Algorithm to perform substring searches.
4. **String Matching in Programming:** In programming, this algorithm is used when a simple and quick pattern matching solution is required.
5. **Bioinformatics:** It can be used to search for DNA or protein sequences in biological data.
6. **Plagiarism Detection:** In plagiarism detection systems, the Naive Algorithm can be used to identify similarities in documents.

While the Naive Algorithm is not the most efficient for large-scale text processing, it provides a basic foundation for understanding and implementing more advanced string-matching algorithms like the Knuth-Morris-Pratt or Boyer-Moore algorithms, which have better time complexities.

**Rabin Karp Algorithm:** The Rabin-Karp Algorithm is designed to improve the efficiency of pattern searching compared to the Naive Algorithm. It employs a hash function to quickly check for matches between the pattern and substrings of the text. The fundamental idea is to compute the hash value of the pattern and the hash values of overlapping substrings of the text. When a hash value match is detected, a character-by-character comparison is performed to confirm the match. This approach reduces the number of character comparisons, making it more efficient for large texts and patterns.



**Algorithm:**

1. Calculate the hash value of the pattern and the first window of text (substring with the same length as the pattern).
2. Slide the window one position at a time from left to right.
3. Check if the hash value of the current window matches the hash value of the pattern.
4. If the hash values match, perform a character-by-character comparison to confirm the match.
5. If a match is confirmed, record the starting position of the match.
6. Calculate the hash value for the next window by using a rolling hash function.
7. Repeat steps 3-6 until you reach the end of the text.
8. Continue sliding the window until you have checked all possible positions.

**Time Complexity:** The time complexity of the Rabin-Karp Algorithm depends on the hash function and how efficiently it can compute hash values. In the worst case, it may need to compare the pattern with all substrings of the text. However, with a good hash function, its average-case time complexity is often better than that of the Naive Algorithm.

In the worst case, when all hash values collide and character-by-character comparisons are required, the time complexity is **O((n - m + 1) \* m)**, where:

* n is the length of the text.
* m is the length of the pattern.

In the average case, with a well-designed hash function, it can achieve better performance, often close to linear time **O(n + m)**.

**Real World Applications:** The Rabin-Karp Algorithm is used in various real-world applications:

1. **Plagiarism Detection:** It's used to find similar or identical passages of text in documents or academic papers.
2. **String Search in Text Editors:** Some text editors use Rabin-Karp for searching and highlighting text.
3. **Data Mining:** It's employed in data mining to search for patterns in large datasets.
4. **Biological Sequence Analysis:** The algorithm can be adapted for searching DNA or protein sequences in bioinformatics.
5. **Content Comparison:** It's used in content comparison tools to find similar sections in documents, websites, or code.
6. **Network Security:** In intrusion detection systems, it can be used to detect known patterns or signatures in network traffic.
7. **Spam Filtering:** Rabin-Karp can be used to identify known spam patterns in email messages.

Overall, the Rabin-Karp Algorithm is a versatile and efficient choice for pattern searching in various domains. Its performance relies heavily on the choice of the hash function and can be optimized for specific use cases.

**Source Code:**

#include <stdio.h>

#include <string.h>

// Function to perform string matching using the Naive algorithm

void naiveStringMatch(char text[], char pattern[]) {

    int m = strlen(text);

    int n = strlen(pattern);

    for (int i = 0; i <= m - n; i++) {

        int j;

        // Check for a match at the current position

        for (j = 0; j < n; j++) {

            if (text[i + j] != pattern[j])

                break;

        }

        // If pattern is found at position i

        if (j == n) {

            printf("Pattern found at position %d\n", i);

        }

    }

}

// Function to perform string matching using the Rabin-Karp algorithm

void rabinKarpStringMatch(char text[], char pattern[], int q) {

    int d = 256; // Number of characters in the input alphabet

    int m = strlen(pattern);

    int n = strlen(text);

    int i, j;

    int p = 0; // Hash value for pattern

    int t = 0; // Hash value for text

    int h = 1;

    // Calculate h = (d^(m-1)) % q

    for (i = 0; i < m - 1; i++) {

        h = (h \* d) % q;

    }

    // Calculate the hash value of the pattern and the first window of text

    for (i = 0; i < m; i++) {

        p = (d \* p + pattern[i]) % q;

        t = (d \* t + text[i]) % q;

    }

    // Slide the pattern over the 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) {

            int match = 1;

            // Check for a match at the current position

            for (j = 0; j < m; j++) {

                if (text[i + j] != pattern[j]) {

                    match = 0;

                    break;

                }

            }

            // If pattern is found at position i

            if (match) {

                printf("Pattern found at position %d\n", i);

            }

        }

        // Calculate the hash value for the next window of text

        if (i < n - m) {

            t = (d \* (t - text[i] \* h) + text[i + m]) % q;

            // Handle negative hash value

            if (t < 0) {

                t = (t + q);

            }

        }

    }

}

int main() {

    char text[100];

    char pattern[100];

    int choice, q;

    printf("Enter the text: ");

    scanf("%s", text);

    printf("Enter the pattern: ");

    scanf("%s", pattern);

    printf("Choose a string matching algorithm:\n");

    printf("1. Naive Algorithm\n");

    printf("2. Rabin-Karp Algorithm\n");

    printf("Enter your choice (1/2): ");

    scanf("%d", &choice);

    switch (choice) {

        case 1:

            naiveStringMatch(text, pattern);

            break;

        case 2:

            printf("Enter a prime number (q) for Rabin-Karp algorithm: ");

            scanf("%d", &q);

            rabinKarpStringMatch(text, pattern, q);

            break;

        default:

            printf("Invalid choice\n");

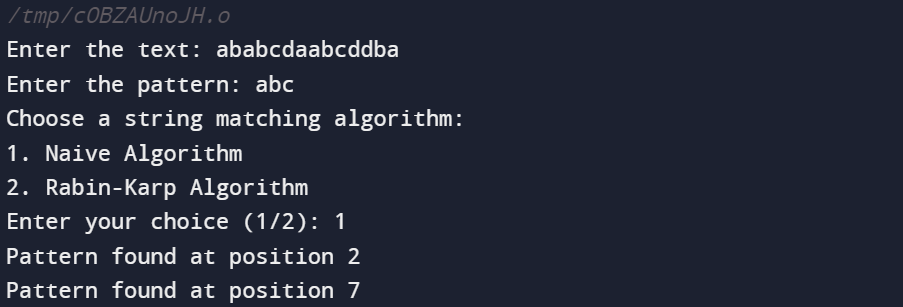
            break;

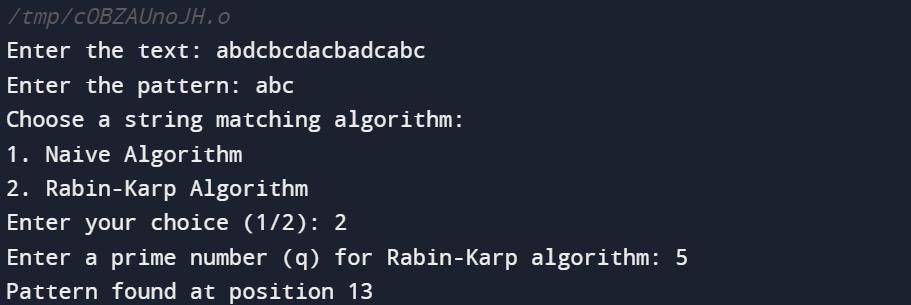
    }

    return 0;

}

**Output:**





**Outcome:** Ability to implement string matching algorithms.

**Conclusion:** In conclusion, the implementation of both the Naive and Rabin-Karp algorithms provides solutions for pattern searching in text. While the Naive Algorithm offers simplicity in its approach, it may not be the most efficient choice for large-scale text processing. On the other hand, the Rabin-Karp Algorithm leverages hashing to improve efficiency, making it suitable for real-world applications with larger datasets. The choice between these algorithms depends on the specific requirements and the characteristics of the text and patterns being searched.