



# EXPERIMENT - 8

## Algorithms Design and Analysis Lab

### Aim

To implement String matching algorithm and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

## EXPERIMENT – 8

### Aim:

To implement String matching algorithm and analyse its time complexity.

### Theory:

String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems. It helps in performing time-efficient tasks in multiple domains. These algorithms are useful in the case of searching a string within another string. String matching is also used in the Database schema, Network systems.

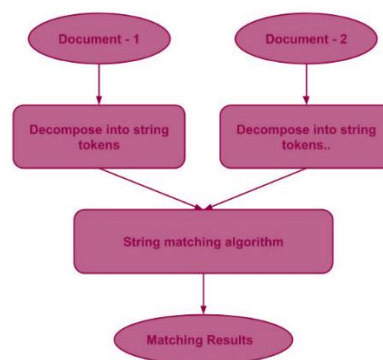
Exact string matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence. These are further classified into four categories:

1. Algorithms based on character comparison:
  - Naive Algorithm: It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
  - KMP (Knuth Morris Pratt) Algorithm: The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.
  - Boyer Moore Algorithm: This algorithm uses best heuristics of Naive and KMP algorithm and starts matching from the last character of the pattern.
  - Using the Trie data structure: It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST.
2. Deterministic Finite Automaton (DFA) method:
  - Automaton Matcher Algorithm: It starts from the first state of the automata and the first character of the text. At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state.
3. Algorithms based on Bit (parallelism method):

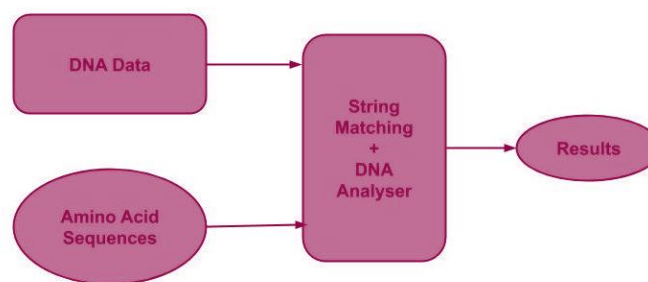
- Aho-Corasick Algorithm: It finds all words in  $O(n + m + z)$  time where  $n$  is the length of text and  $m$  be the total number characters in all words and  $z$  is total number of occurrences of words in text. This algorithm forms the basis of the original Unix command `fgrep`.
4. Hashing-string matching algorithms:
- Rabin Karp Algorithm: It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

### Applications of String Matching Algorithms:

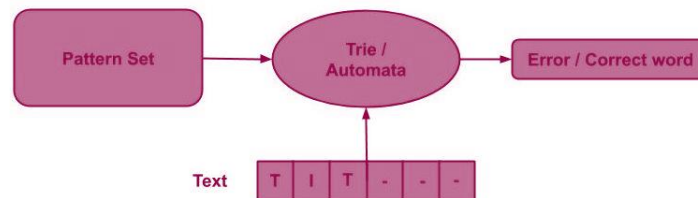
- **Plagiarism Detection:** The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.



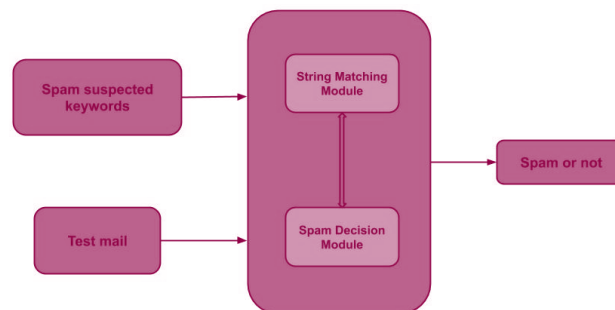
- **Bioinformatics and DNA Sequencing:** Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.



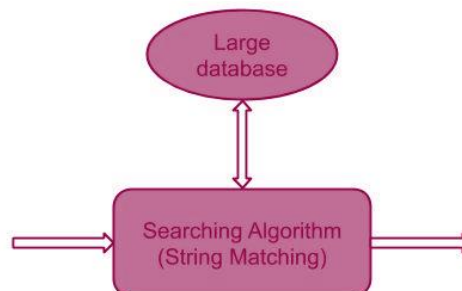
- **Digital Forensics:** String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.
- **Spelling Checker:** Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.



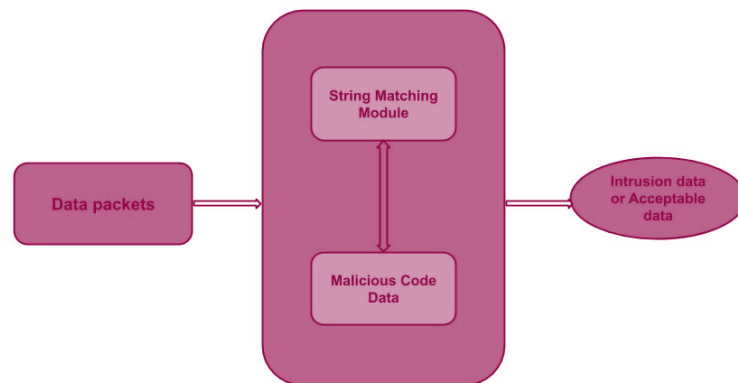
- **Spam filters:** Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.



- **Search engines or content search in large databases:** To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.



- **Intrusion Detection System:** The data packets containing intrusion-related keywords are found by applying string matching algorithms. All the malicious code is stored in the database, and every incoming data is compared with stored data. If a match is found, then the alarm is generated. It is based on exact string matching algorithms where each intruded packet must be detected.



## NAÏVE STRING-MATCHING ALGORITHM

### Source Code:

```

#include <bits/stdc++.h>
using namespace std;

void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;
    }
}
  
```

```
        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            cout << "Pattern found at index "
                << i << endl;
    }
}

int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    search(pat, txt);

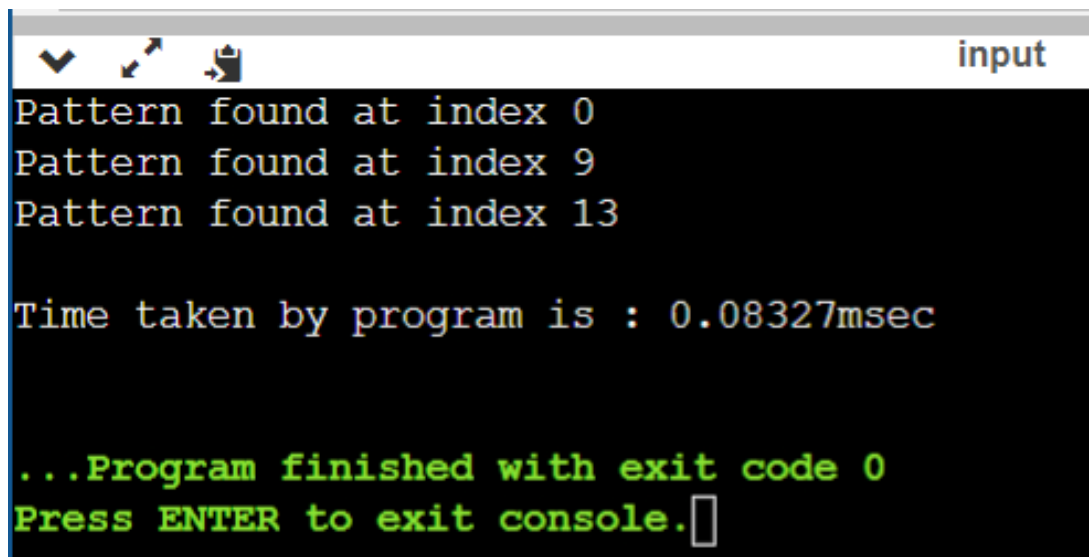
    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

### Output:



```
input
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

Time taken by program is : 0.08327msec

...Program finished with exit code 0
Press ENTER to exit console.█
```

## Rabin Karp String Matching Algorithm

Rabin Karp algorithm needs to calculate hash values for following strings.

- 1) Pattern itself.
- 2) All the substrings of the text of length  $m$ .

Since we need to efficiently calculate hash values for all the substrings of size  $m$  of text, we must have a hash function which has the following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say  $hash(txt[s+1 .. s+m])$  must be efficiently computable from  $hash(txt[s .. s+m-1])$  and  $txt[s+m]$  i.e.,  $hash(txt[s+1 .. s+m]) = rehash(txt[s+m], hash(txt[s .. s+m-1]))$  and rehash must be  $O(1)$  operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is the numeric value of a string.

For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value.

### Source Code:

```
#include <bits/stdc++.h>
using namespace std;

#define d 256

void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // 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 + pat[i]) % q;
    t = (d * t + txt[i]) % q;
}

// Slide the pattern over text one by one
for (i = 0; i <= N - M; i++)
{
    if (p == t)
    {
        bool flag = true;
        /* Check for characters one by one */
        for (j = 0; j < M; j++)
        {
            if (txt[i + j] != pat[j])
            {
                flag = false;
                break;
            }
            if (flag)
                cout << i << " ";
        }

        // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

        if (j == M)
            cout << "Pattern found at index " << i << endl;
    }

    // Calculate hash value for next window of text: Remove leading digit,
    add trailing digit
    if (i < N - M)
    {
        t = (d * (t - txt[i] * h) + txt[i + M]) % q;

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

}

int main()
{
    char txt[] = "AABAACAADAABAAABAA";
```



```
char pat[] = "AABA";

// A prime number
int q = 101;

auto start = chrono::high_resolution_clock::now();
ios_base::sync_with_stdio(false);

search(pat, txt, q);

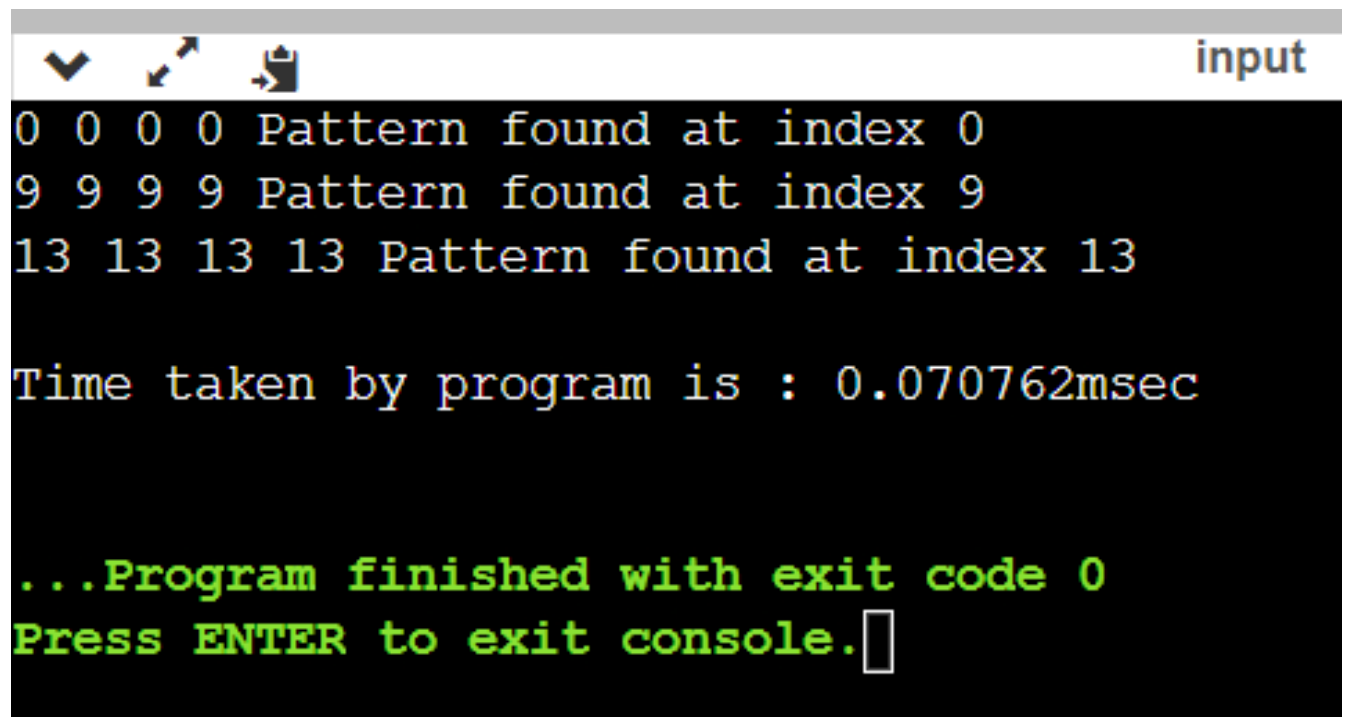
auto end = chrono::high_resolution_clock::now();
double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

time_taken *= 1e-9 * 1000;

cout << "\nTime taken by program is : " << time_taken << setprecision(6);
cout << "msec" << endl;

return 0;
}
```

### Output:



```
0 0 0 0 Pattern found at index 0
9 9 9 9 Pattern found at index 9
13 13 13 13 Pattern found at index 13

Time taken by program is : 0.070762msec

...Program finished with exit code 0
Press ENTER to exit console.█
```

## Knuth Morris Pratt algorithm

- KMP algorithm preprocesses `pat[]` and constructs an auxiliary **`lps[]`** of size `m` (same as size of pattern) which is used to skip characters while matching.
- **name `lps` indicates longest proper prefix which is also suffix..** A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- We search for `lps` in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern `pat[0..i]` where `i = 0 to m-1`, `lps[i]` stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.
- `lps[i] = the longest proper prefix of pat[0..i]`  
which is also a suffix of `pat[0..i]`.

### Matching Overview

`txt = "AAAAABAAABA"`

`pat = "AAAA"`

We compare first window of **`txt`** with **`pat`**

`txt = "AAAAABAAABA"`

`pat = "AAAA"` [Initial position]

We find a match. This is same as [Naive String Matching](#).

In the next step, we compare next window of **`txt`** with **`pat`**.

`txt = "AAAAABAAABA"`

`pat = "AAAA"` [Pattern shifted one position]

This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

## Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array `lps[]` that tells us the count of characters to be skipped.

## Source Code:

```
#include <bits/stdc++.h>
using namespace std;

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
```

```
        else if (i < N && pat[j] != txt[i])
        {
            // Do not match lps[0..lps[j-1]] characters, they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char *pat, int M, int *lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int main()
```

```
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    KMPSearch(pat, txt);

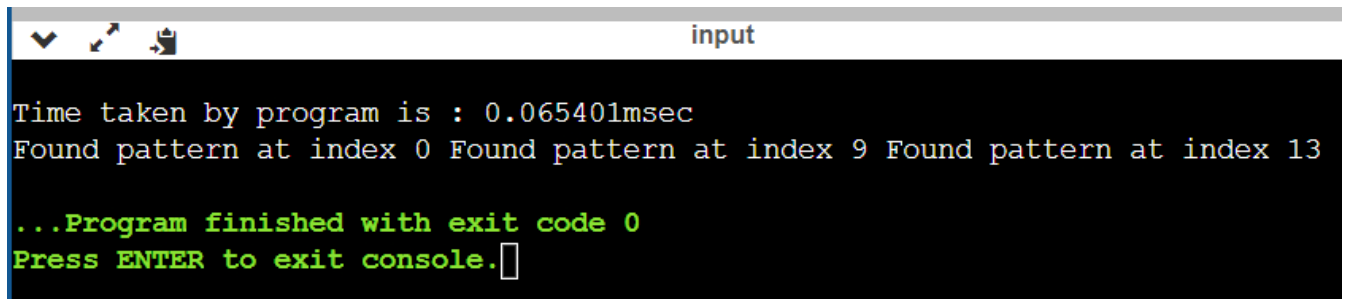
    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

### Output:



```
Time taken by program is : 0.065401msec
Found pattern at index 0 Found pattern at index 9 Found pattern at index 13

...Program finished with exit code 0
Press ENTER to exit console.
```

## Viva Questions

**1. What is the worst case time complexity of KMP algorithm for pattern searching ( $m$  = length of text,  $n$  = length of pattern)?**

Ans.

KMP algorithm is an efficient pattern searching algorithm. It has a time complexity of  $O(m)$  where  $m$  is the length of text.

**2. The naive pattern searching algorithm is an in place algorithm or not.**

Ans.

The auxiliary space complexity required by naive pattern searching algorithm is  $O(1)$ . So it qualifies as an in place algorithm.

**3. Describe about Rabin Karp algorithm and naive pattern searching algorithm worst case time complexity.**

Ans.

The worst case time complexity of Rabin Karp algorithm is  $O(m*n)$  but it has a linear average case time complexity. So Rabin Karp and naive pattern searching algorithm have the same worst case time complexity.

**4. What is a Rabin and Karp Algorithm?**

Ans.

The string matching algorithm which was proposed by Rabin and Karp, generalizes to other algorithms and for two-dimensional pattern matching problems.

**5. What is the pre-processing time of Rabin and Karp Algorithm?**

Ans.

The for loop in the pre-processing algorithm runs for  $m$  (length of the pattern) times. Hence the pre-processing time is  $\Theta(m)$ .

**6. What is the basic formula applied in Rabin Karp Algorithm to get the computation time as  $\Theta(m)$ ?**

Ans.

The pattern can be evaluated in time  $\Theta(m)$  using Horner's rule:  
$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10P[1]) \dots)).$$

**7. What happens when the modulo value( $q$ ) is taken large?**

Ans.

If the modulo value( $q$ ) is large enough then the spurious hits occur infrequently enough that the cost of extra checking is low.