



# EXTERNAL PRACTICAL EXPERIMENT

Algorithms Design and Analysis Lab: ETCS-351

## Aim

To implement Knuth Morris Pratt algorithm and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

## EXPERIMENT – 1

### Aim:

To implement Knuth Morris Pratt algorithm and analyse its time complexity.

### Theory:

String matching algorithms 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.

### 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

`text = "AAAAABAAABA"`

`pattern = "AAAA"`

We compare first window of **text** with **pattern**

`text = "AAAAABAAABA"`

`pattern = "AAAA" [Initial position]`

We find a match.

In the next step, we compare next window of **text** with **pattern**.

text = "AAAAABAAABA"

pattern = "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.

### Source Code:

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

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

    if (M > N) {
        cout << "Pattern can't be longer than the String!" << endl;
        return -1;
    }

    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[]
    bool match = 0;
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            cout << "Found pattern at index: " << (i - j) << endl;
            j = lps[j - 1];
            match = 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;
    }
}
if (!match)
{
    cout << "No Match Found!" << endl;
    return 0;
}
return 1;
}

int main()
{
    // char text[] = "AABAACAADAABAAABAA";
    // char pattern[] = "AABA";

    char text[100], pattern[100];

    cout << "Enter the string for pattern matching: ";
    cin >> text;
    cout << "Enter the pattern that you need to check: ";
    cin >> pattern;

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

    KMPSearch(pattern, text);







    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 KMP program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

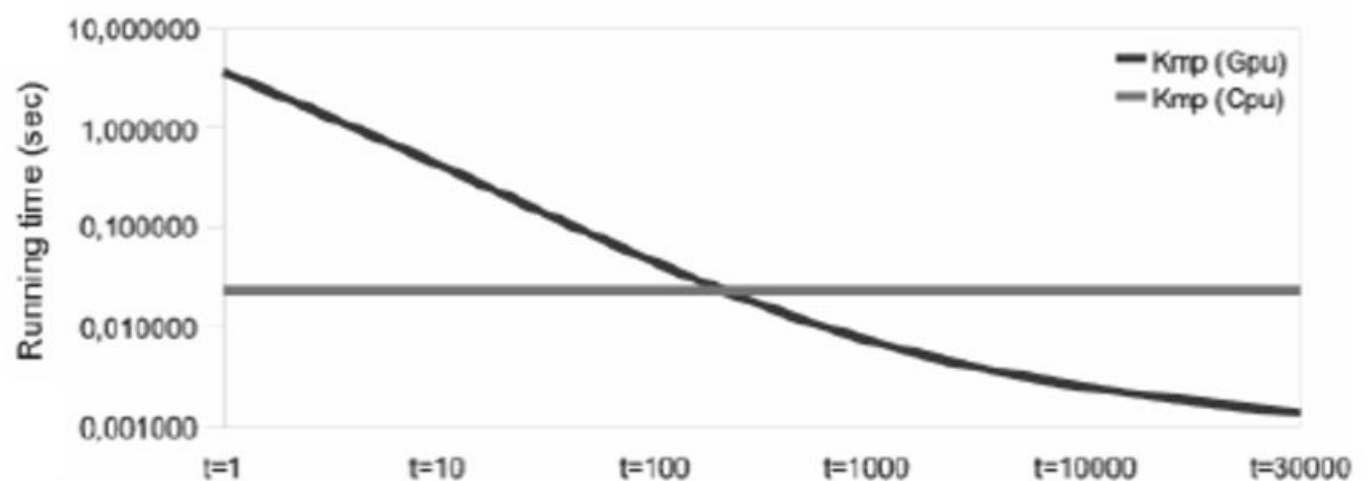
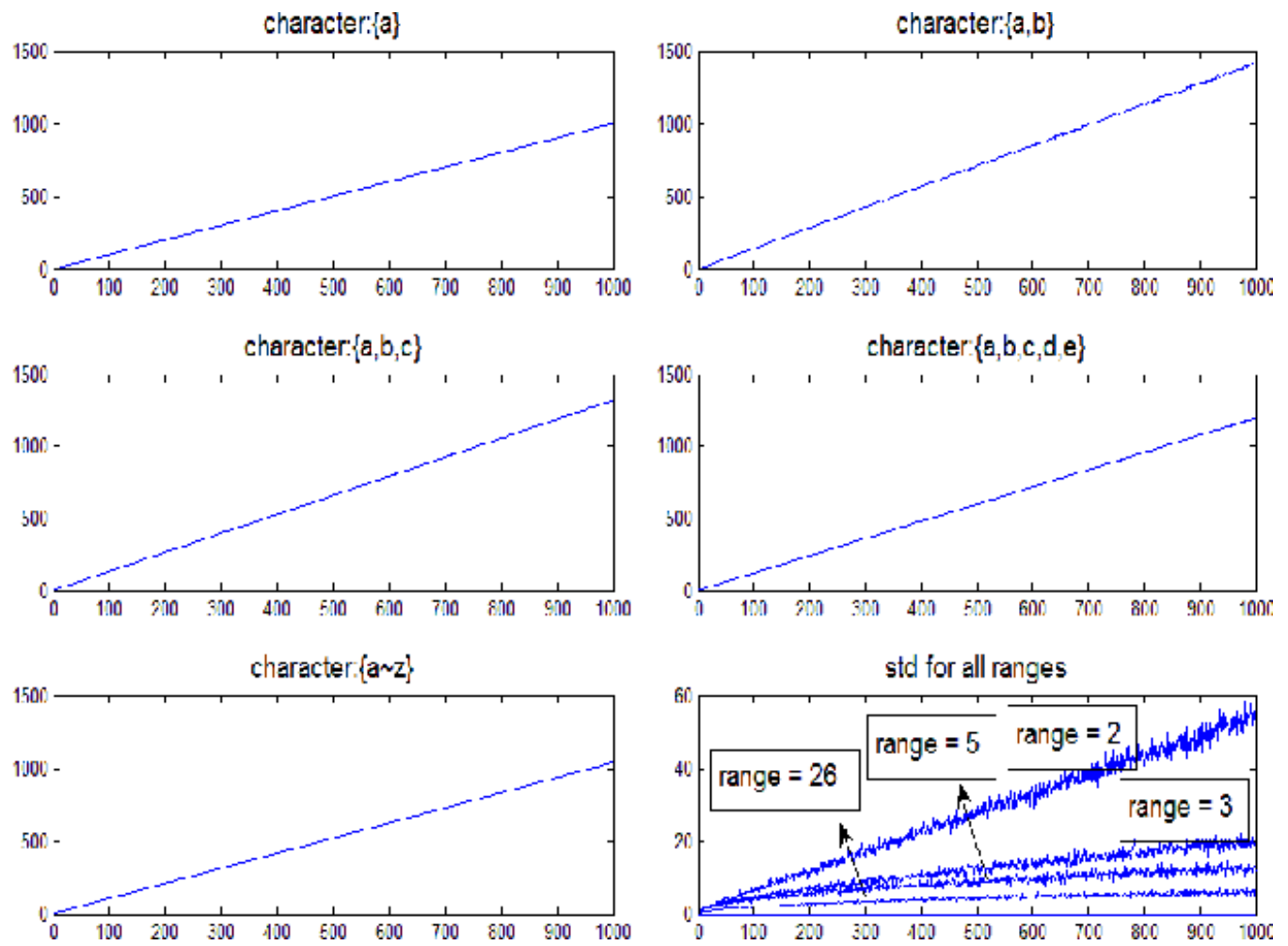
    return 0;
}
```

## Output:

   input  
Enter the string for pattern matching: AABAACAADAABAAABAA  
Enter the pattern that you need to check: AABA  
Found pattern at index: 0  
Found pattern at index: 9  
Found pattern at index: 13  
  
Time taken by KMP program is : 0.103687msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.    input  
Enter the string for pattern matching: AAAAABAAABA  
Enter the pattern that you need to check: AAAA  
Found pattern at index: 0  
Found pattern at index: 1  
  
Time taken by KMP program is : 0.11628msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.

input  
Enter the string for pattern matching: AAAAAA  
Enter the pattern that you need to check: BBB  
No Match Found!  
  
Time taken by KMP program is : 0.087408msec  
  
...Program finished with exit code 0  
Press ENTER to exit console. input  
Enter the string for pattern matching: AAAAAA  
Enter the pattern that you need to check: AAAAAAAAAA  
Pattern can't be longer than the String!  
  
Time taken by KMP program is : 0.062987msec  
  
...Program finished with exit code 0  
Press ENTER to exit console.

## Time Complexity:





## 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?

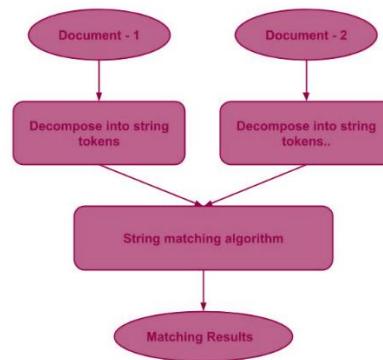
Ans.

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

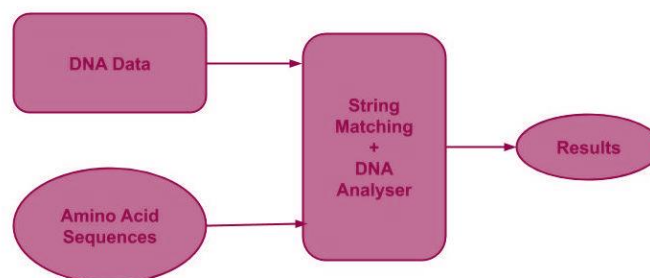
### Applications of String Matching Algorithms:

- **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.

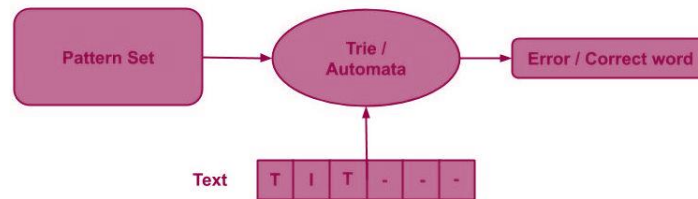


- **Bioinformatics and DNA Sequencing:** Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.

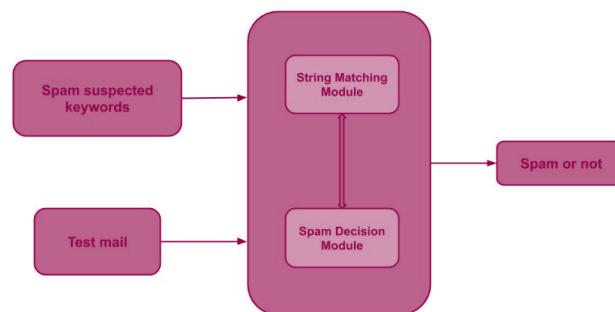


- **Digital Forensics:** String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.
- **Spelling Checker:** Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The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

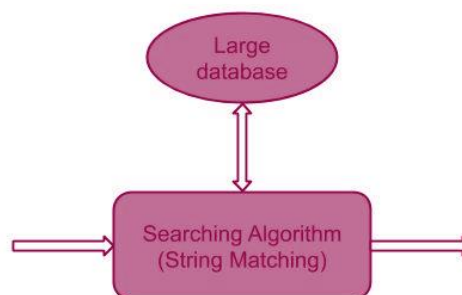
such pattern occurs, it is shown by reaching the acceptance state.



- 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.

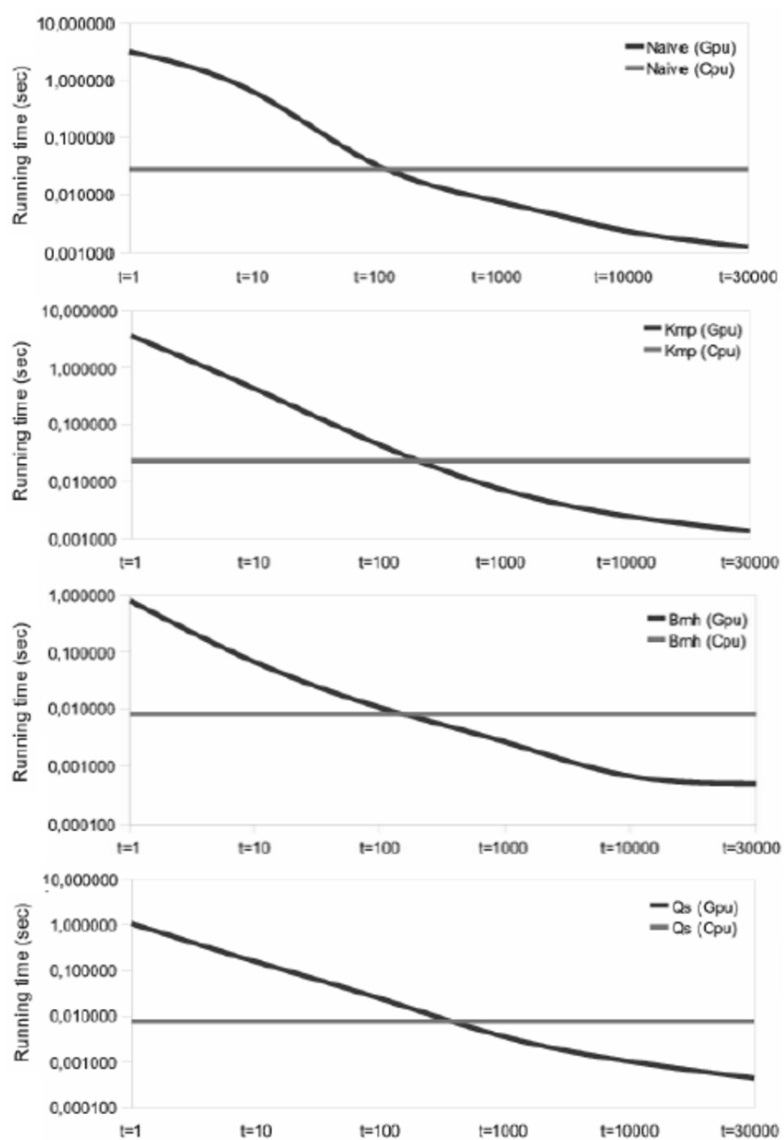
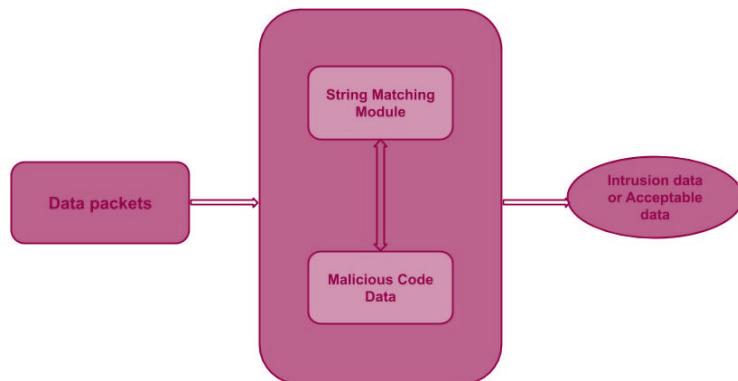


- Search engines or content search in large databases:** To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.



- 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.



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.