

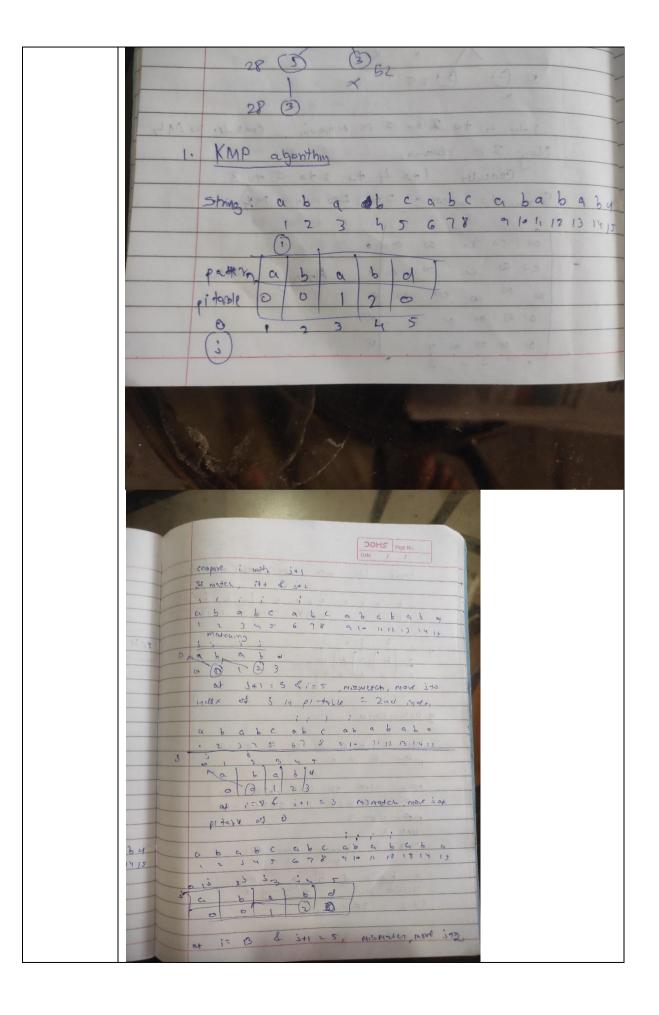
Bharatiya Vidya Bhavan's SARDAR PATEL INSTITUTE OF TECHNOLOGY

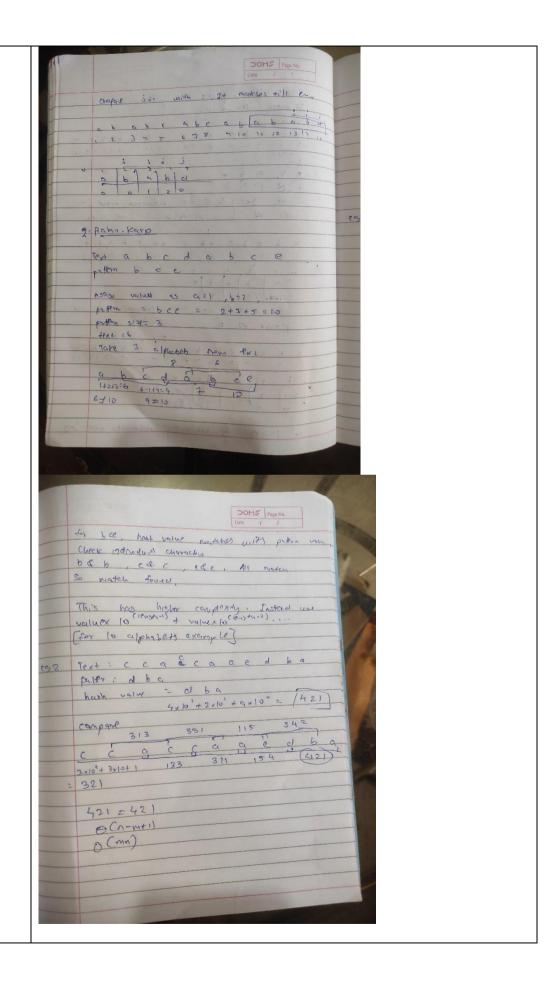
(Autonomous Institute Affiliated to University of Mumbai) Munshi Nagar, Andheri (W), Mumbai – 400 058. Department of MCA

Experiment	10
Aim	To understand and implement String Matching Algorithm
Objective	1) Write Pseudocode for any 2-string matching algorithm
	2) Implementing the above mentioned 2 string matching
	algorithms
	3) Calculating time complexity of the given problems
	4) Solve the string matching for both the algorithm on pen and
	paper
Name	Vivek Tiwari
UCID	2023510059
Class	FYMCA
Batch	С

```
KMP
Algorithm
and
               KMP(text, pattern):
                 compute_lps(pattern)
Explanatio
n of the
                 i = 0
technique
                 j = 0
used
                 while i < text.length():
                    if pattern[j] == text[i]:
                       i++
                       j++
                       if j == pattern.length():
                         return i - j
                    else:
                       if j != 0:
                         j = lps[j-1]
                       else:
                         i++
                 return -1
               compute_lps(pattern):
                 lps[0] = 0
                 len = 0
                 i = 1
                 while i < pattern.length():
                    if pattern[i] == pattern[len]:
```

```
len++
       lps[i] = len
       i++
     else:
       if len != 0:
          len = lps[len - 1]
       else:
          lps[i] = 0
          i++
Rabin Karp
RabinKarp(text, pattern):
  pattern_hash = hash(pattern)
  text_hash = hash(text[0:pattern.length()])
  for i from 0 to text.length() - pattern.length():
     if pattern_hash == text_hash and text[i:i+pattern.length()] == pattern:
       return i
     text_hash = recompute_hash(text_hash, text[i], text[i+pattern.length()])
  return -1
hash(str):
  hash_value = 0
  for i from 0 to str.length()-1:
     hash_value = (hash_value * 256 + str[i]) % prime
  return hash_value
recompute_hash(old_hash, old_char, new_char):
  hash_value = (old_hash - old_char * (256^(pattern.length()-1))) * 256 +
new_char
  return hash_value % prime
```





```
1. KMP
Program(C
ode)
            #include <iostream>
            #include <vector>
            #include <string>
            using namespace std;
            vector<int> computeLPS(const string& pattern) {
                 int m = pattern.size();
                 vector<int> lps(m);
                 lps[0] = 0;
                 int len = 0;
                 int i = 1;
                while (i < m) {
                     if (pattern[i] == pattern[len]) {
                         len++;
                         lps[i] = len;
                         i++;
                     } else {
                         if (len != 0) {
                             len = lps[len - 1];
                         } else {
                             lps[i] = 0;
                             i++;
                 return lps;
            void KMP(const string& text, const string& pattern) {
                 int n = text.size();
                 int m = pattern.size();
                 vector<int> lps = computeLPS(pattern);
                 int i = 0, j = 0;
                while (i < n) {
                     if (pattern[j] == text[i]) {
                         i++;
                         j++;
                     if (j == m) {
                         cout << "Pattern found at index " << i - j << endl;</pre>
                         j = lps[j - 1];
                     } else if (i < n && pattern[j] != text[i]) {</pre>
                         if (j != 0) {
                             j = lps[j - 1];
```

} else {
 i++;

```
if (j != m)
        cout << "Pattern not found" << endl;</pre>
int main() {
    string text, pattern;
    cout << "Enter the text: ";</pre>
    getline(cin, text);
    cout << "Enter the pattern: ";</pre>
    getline(cin, pattern);
   KMP(text, pattern);
    return 0;
   2. Rabin Karp
#include <iostream>
#include <string>
#include <cmath>
using namespace std;
const int prime = 101;
int hash_string(const string& str, int len) {
    int hash_value = 0;
    for (int i = 0; i < len; ++i) {
        hash_value += str[i] * static_cast<int>(pow(256, len -
i - 1));
        hash_value %= prime;
    return hash value;
int recompute_hash(int old_hash, char old_char, char new_char,
int pattern_length) {
    int hash_value = (old_hash - old_char *
static_cast<int>(pow(256, pattern_length - 1))) * 256 +
new_char;
    hash_value %= prime;
    return (hash_value + prime) % prime;
int rabin_karp(const string& text, const string& pattern) {
    int text_length = text.length();
```

```
int pattern length = pattern.length();
                int pattern_hash = hash_string(pattern, pattern_length);
                int text hash = hash string(text.substr(0, pattern length),
            pattern length);
                for (int i = 0; i <= text length - pattern_length; ++i) {</pre>
                    if (text hash == pattern hash && text.substr(i, pat-
            tern_length) == pattern) {
                        return i;
                    if (i < text length - pattern length) {</pre>
                        text_hash = recompute_hash(text_hash, text[i],
            text[i + pattern length], pattern length);
                return -1;
            int main() {
                string text, pattern;
                cout << "Enter the text: ";</pre>
                getline(cin, text);
                cout << "Enter the pattern to search for: ";</pre>
                getline(cin, pattern);
                int index = rabin karp(text, pattern);
                if (index != -1) {
                    cout << "Pattern found at index: " << index << endl;</pre>
                } else {
                    cout << "Pattern not found in the text." << endl;</pre>
                return 0;
               1. KMP
Output
                   Enter the text: ababcabcabababd
                   Enter the pattern: ababd
                   Pattern found at index 10
                   Pattern not found
               2. Rabin Karp
                   Enter the text: ccaccaaedba
                   Enter the pattern: dba
                   Pattern found at index 8
                   Pattern not found
```

Justificatio n of the complexity calculated

The Knuth-Morris-Pratt (KMP) algorithm has a time complexity of O(n + m), where n is the length of the text and m is the length of the pattern. This complexity arises from two main factors: the linear time complexity of preprocessing the pattern to compute the longest prefix suffix (LPS) array, and the linear time complexity of traversing the text while matching the pattern. The LPS array preprocessing step takes O(m) time, and the pattern matching step takes O(n) time in the worst case, resulting in a total time complexity of O(n + m). The Rabin-Karp algorithm, on the other hand, has an average-case time complexity of O(n + m) and a worst-case time complexity of O(n * m). This complexity arises from the rolling hash function used to efficiently compare the hash values of the pattern and substrings of the text. In the average case, where the hash function distributes hash values evenly, the algorithm achieves linear time complexity, similar to the KMP algorithm. However, in the worst case, where multiple substrings have the same hash value as the pattern, the algorithm may need to compare the pattern character by character with each substring, resulting in a time complexity of O(n * m).

Conclusion

The Knuth-Morris-Pratt (KMP) algorithm offers advantages in scenarios where the pattern remains constant while searching through different texts, as it preprocesses the pattern to efficiently skip unnecessary comparisons during the search process. This makes it particularly suitable for applications such as text editors, search engines, and DNA sequencing, where pattern matching is performed repeatedly. Additionally, KMP's linear time complexity makes it efficient for large datasets and long patterns. On the other hand, the Rabin-Karp algorithm provides advantages in scenarios where the pattern may change frequently or when multiple patterns need to be searched simultaneously within a single text. Its ability to compare hash values allows for efficient pattern matching, and its averagecase time complexity makes it suitable for applications like plagiarism detection, spell checkers, and string similarity comparison. However, Rabin-Karp's worst-case time complexity may be less favorable compared to KMP, especially for highly repetitive patterns or texts. Overall, the choice between the two algorithms depends on the specific requirements and characteristics of the problem at hand.