Name : Sonal Dilip Gholap

UID : 2021600020

Branch : CSE-AIML

Batch : B

Experiment : 10

Aim : Experiment based on String Matching Algorithm (Knuth Morris Pratt)

- **Objectives :**

1. To study KMP string matching algorithm.
2. To implement KMP (Knuth Morris Pratt) algorithm.
3. To derive and analyze the time complexity of KMP algorithm.

- **Theory :**

1. String matching algorithms are a set of techniques used to find patterns or sequences of characters within a given text or string. These algorithms are widely used in computer science and are an essential part of text processing, data mining, and information retrieval systems.
2. The Knuth-Morris-Pratt (KMP) algorithm is a linear time string matching algorithm that searches for occurrences of a pattern in a text.
3. The KMP algorithm preprocesses the pattern to create a partial match table, also known as the failure function or the border array. This table stores information about the longest proper prefix of the pattern that is also a suffix of the substring ending at each position.
4. The algorithm then scans the text from left to right, maintaining two pointers, one pointing to the current position in the pattern and the other pointing to the current position in the text. If the characters match, both pointers are incremented. If the characters do not match, the algorithm uses the partial match table to skip over unnecessary comparisons.
5. The KMP algorithm requires O(m) extra space to store the partial match table, where m is the length of the pattern. This can be a disadvantage for very long patterns. The algorithm also has a relatively high overhead for short patterns, since the table construction dominates the running time.

- **Why is KMP better than Naive solution?**

1. Time Complexity : The brute force algorithm compares each character of the pattern to the corresponding character of the text, potentially multiple times. This leads to a time complexity of O(mn). In contrast, the KMP algorithm compares each character of the pattern to the corresponding character of the text at most once, and the partial match table can be computed in O(m) time. This leads to a time complexity of O(m+n), which is much faster than O(mn) for large datasets.
2. Efficiency : The KMP algorithm skips over unnecessary comparisons using the partial match table, which can greatly reduce the number of comparisons required. In contrast, the brute force algorithm compares every possible substring of the text to the pattern, which can lead to a large number of unnecessary comparisons.
3. Multiple Pattern Matching : The KMP algorithm can be used to match multiple patterns to a single text, whereas the brute force algorithm requires a separate search for each pattern. This can be a significant advantage in certain applications, such as text search engines.

- **Pseudocode for KMP**

```
ComputeLPS()

    i ← 0
    len ← 0
    m ← pattern.length
    LPS[0] = 0

    for i = 1 to m do
       if pattern[i] = pattern[len] then
          len = len + 1
          LPS[i] = len
       else
          if len != 0
             len ← LPS[len - 1]
            i = i - 1
          else
             LPS[i] = 0
```

```
KMP()
    i ← 0
    j ← 0
        while i < n:
            if pat[j] == s[i]:
                i = i + 1
                j = j + 1
                if j == m:
                    pattern found at index (i-j)
            else:
                if j != 0:
                    j = LPS[j - 1]
                else:
                    i = i + 1
```

Derivation of Time Complexity

Page No. _____

* KMP

Preprocessing i.e. Computing LPS array — Time

```
n = string size                                   1
m = pattern size                                  1
len = 0                                           1
i = 0                                             1
LPS[0] = 0                                        1
                                                  1
for i = 1 to m do                                 m
    if pat[i] = pat[len] then                     m
        len += 1                                  m
        LPS[i] = len                              m
    else                                          m
        if len != 0                               m
            len = LPS[len-1]                      m
            i = i-1                               m
        else                                      m
            LPS[i] = 0                            m
```

$\therefore T(n) = m + m + m + m + m + m + m + m + m + m$

$= 10m$

$= O(m)$

Time complexity of preprocessing = $O(m)$

```
KMP( )                                  Time
i=0                                      1
j=0                                      1
while i<n                                n
    if pat[j] = s[i]                     n
        i++                              n
        j++                              n
        if j=m then                      n
            Pattern Found                n
    else                                 n
        if j!=0                          n
            j = LPS[j-1]                 n
        else                             n
            i++                          n
```

$\therefore T(n) = 1+1+11n$

$T(n) = O(n)$

$\therefore$ Time complexity of KMP = Preprocessing + String matching

$T(n) = O(m+n)$

Code :

```
#include<stdio.h>
#include<vector>
#include<bits/stdc++.h>
#include<iterator>
#include<iostream>

using namespace std;

int main(){

    string s;
    cout<<"Enter string: ";
    getline(cin,s);
    string pat;
    cout<<"Enter pattern to be searched: ";
    getline(cin,pat);

    int m = pat.size();
    int n = s.size();

    vector<int> lps(m);
    lps[0]=0;
    int len=0;

    //computing LPS array
    for(int i=1;i<m;i++){
        if(pat[i]==pat[len]){
            len++;
            lps[i]=len;
        }
        else{
            if(len!=0) {
                len = lps[len-1];
                i--;
            }
            else{
                lps[i] = 0;
            }
        }
    }

    int i=0;
    int j=0;

    //KMP
    while(i<n){
        if(pat[j]==s[i]){   //if it is a match
            i++;
```

```
            j++;

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

        if(i<n && pat[j]!=s[i]){        //if it's not a match
            if(j!=0){
                j = lps[j-1];
            }
            else{       //if you reach the start of the pattern
                i++;
            }
        }
    }

    return 0;
}
```

Output :

- Test Case 1 –

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.3-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-xngo5grq.h2c' '--stdout=Microsoft-MIEngine-Out-yx4ncrzs.wzg' '--stderr=Microsoft-MIEngine-Err
or-3elybzr0.fes' '--pid=Microsoft-MIEngine-Pid-k404hlzp.mri' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter string: this is a test text
Enter pattern to be searched: test
Pattern found at index 10
PS C:\C++ Learning Course>
```

- Test Case 2 –

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.3-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-wxc10ger.u4g' '--stdout=Microsoft-MIEngine-Out-mpuiaeyw.x3n' '--stderr=Microsoft-MIEngine-Err
or-n0gnxseo.3dh' '--pid=Microsoft-MIEngine-Pid-20k4no3m.n5j' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter string: aabaacaadaabaaba
Enter pattern to be searched: aaba
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
PS C:\C++ Learning Course>
```

Result :

1. Time complexity of KMP algorithm – **O(m+n)**
2. Space complexity of KMP algorithm – **O(m)**

Conclusion :

1. The Knuth-Morris-Pratt (KMP) algorithm is a highly efficient string searching algorithm that can significantly reduce the time complexity of searching for a pattern within a larger text string.
2. KMP algorithm outperforms naive string searching algorithm in terms of time complexity and efficiency.
3. The KMP algorithm avoids unnecessary comparisons by utilizing information from previous comparisons using a partial match table.