Experiment No. 5

Title: To study optimization of String Searching problem using KMP algorithm

Batch: B-4 **Roll No: 16010422234** **Name: Chandana Ramesh Galgali**

Experiment No.: 5

Aim: To study optimization of string searching problem using KMP Algorithm

Resources needed: Text Editor, C/C++ IDE

Theory:

Pattern searching is an important problem in computer science. When we search for a string in a notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

The naive method doesn't work well in cases where we see many matching characters followed by a mismatching character.

Consider following example,

text = "aaaaaaab"

pattern = "aaab"

Here if there are 'n' letters in text and ' m' letters in pattern then the time complexity of Naive method/Brute Force method is O(m(n–m+1)) where n is very large as compared to m. Hence the time complexity can be considered as O(nm).

This is the drawback of the Naïve Method. So to overcome this problem the KMP method was introduced.

KMP Algorithm is one of the most popular pattern matching algorithms. KMP stands for Knuth Morris Pratt. The KMP algorithm was invented by Donald Knuth and Vaughan Pratt together and independently by James H Morris in the year 1970. In the year 1977, all the three jointly published the KMP Algorithm.

The KMP algorithm was the first linear time complexity algorithm for string matching. The KMP algorithm is one of the string matching algorithms used to find a "Pattern" in a "Text".

LPS Table (Longest proper Prefix which is also Suffix)

This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "Prefix Table" to skip characters comparison while matching. Sometimes the prefix table is also known as LPS Table. Here LPS stands for "Longest proper Prefix which is also Suffix".

Steps for creating LPS Table

Step 1 – Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])

Step 2 – Define variables i and j. Set i = 0, j = 1 and LPS[0] = 0

Step 3 – Compare the characters at Pattern[i] and Pattern[j].

Step 4 – If both are matched, then
LPS[j] = i+1
i = i+1
j = j+1 and Goto to Step 3

Step 5 – If both are not matched then check the value of variable 'i'.
Case a) If it is '0' then set LPS[j] = 0 and increment 'j' value by one
Case b) If it is not '0' then set i = LPS[i-1].
Goto Step 3

Step 6 – Repeat above steps until all the values of LPS[ ] are filled

Using LPS Table for searching Pattern in Text
We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred. When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

Steps for searching Pattern in Text using LPS table

Step 1 – Initialize i and j pointers as i = 0 and j = 0. I pointer is used to iterate over characters ‘n’ in “Text” and j pointer is used to iterate over characters ‘m’ in “Pattern”

Step 2 – Compare Text[i] and Pattern [j]

Step 3 – If matched, then i = i+1, j = j+1 and Goto Step 2

Step 4 – If not matched, then check value of j

Case a) If j != 0, then j = lps[j–1]

Case b) If j = 0, then i = i+1

Goto step 2

Step 4 – If j == m, then pattern found at (i–j), j = lps[j–1] and goto Step 2

Step 5 – Repeat above steps until i < n–m+1

Activity:

Given 2 strings, P and T, find the number of occurrences of P in T.

Input format
First line contains string P, and the second line contains the string T.

Output format
Print a single integer, the number of occurrences of P in T.

Constraints
$1 \leq |P| \leq |T| \leq 10^5$

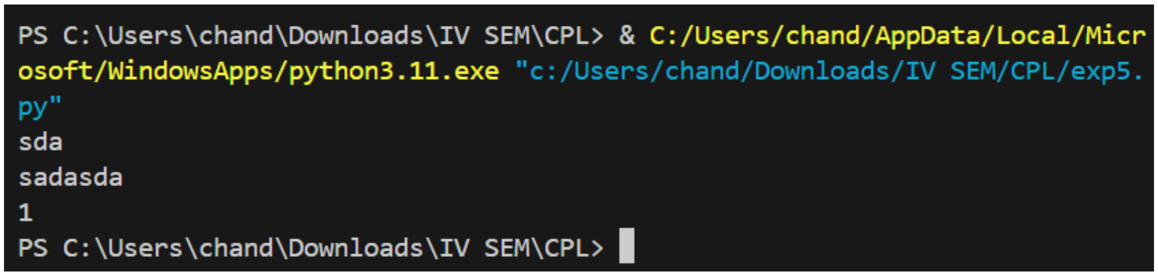
Sample Input
sda
sadasda

Sample Output
1

Program:

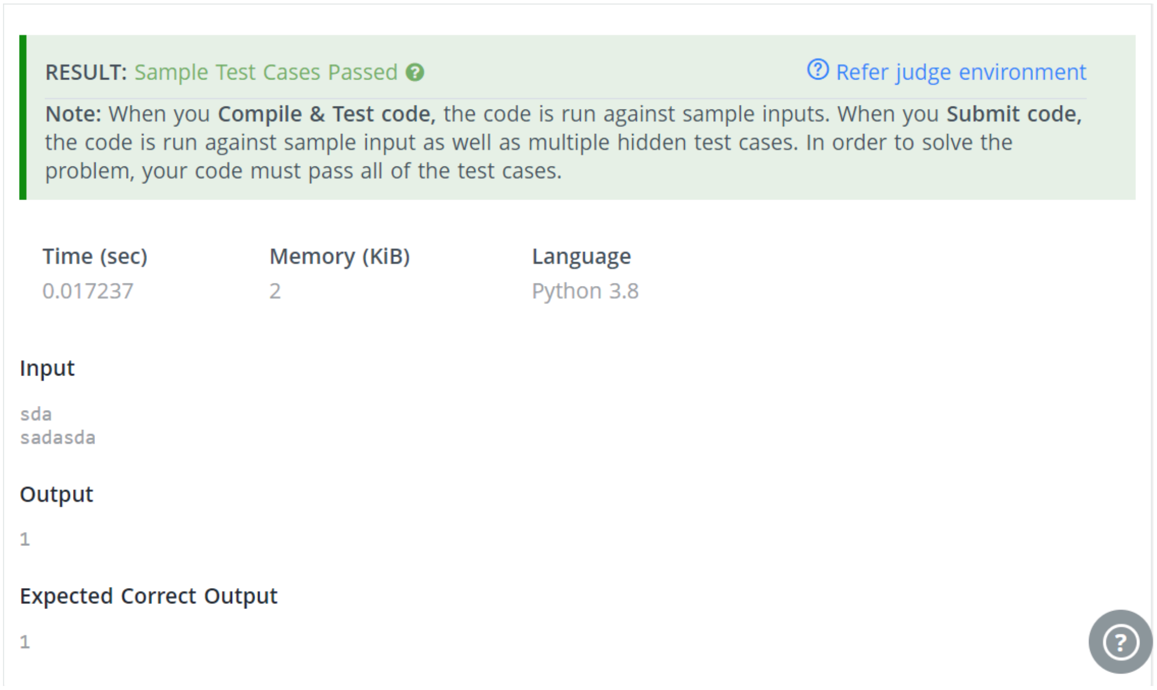
```python
def computeLPSArray(P):
    m = len(P)
    lps = [0] * m
    l = 0
    i = 1
    while i < m:
        if P[i] == P[l]:
            l += 1
            lps[i] = l
            i += 1
        else:
            if l != 0:
                l = lps[l-1]
            else:
                lps[i] = 0
                i += 1
    return lps

def KMPSearch(P, T):
    M = len(P)
    N = len(T)
    lps = computeLPSArray(P)
    i = j = 0
    count = 0
    while i < N:
        if P[j] == T[i]:
            i += 1
            j += 1
        if j == M:
            count += 1
            j = lps[j-1]
        elif i < N and P[j] != T[i]:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
    return count

P = input()
T = input()

result = KMPSearch(P, T)
print(result)
```

Output:



```
PS C:\Users\chand\Downloads\IV SEM\CPL> & C:/Users/chand/AppData/Local/Micr
osoft/WindowsApps/python3.11.exe "c:/Users/chand/Downloads/IV SEM/CPL/exp5.
py"
sda
sadasda
1
PS C:\Users\chand\Downloads\IV SEM\CPL> 
```

Test Result:



RESULT: Sample Test Cases Passed

Refer judge environment

Note: When you **Compile & Test code,** the code is run against sample inputs. When you **Submit code,** the code is run against sample input as well as multiple hidden test cases. In order to solve the problem, your code must pass all of the test cases.

Time (sec)	Memory (KiB)	Language
0.017237	2	Python 3.8

Input

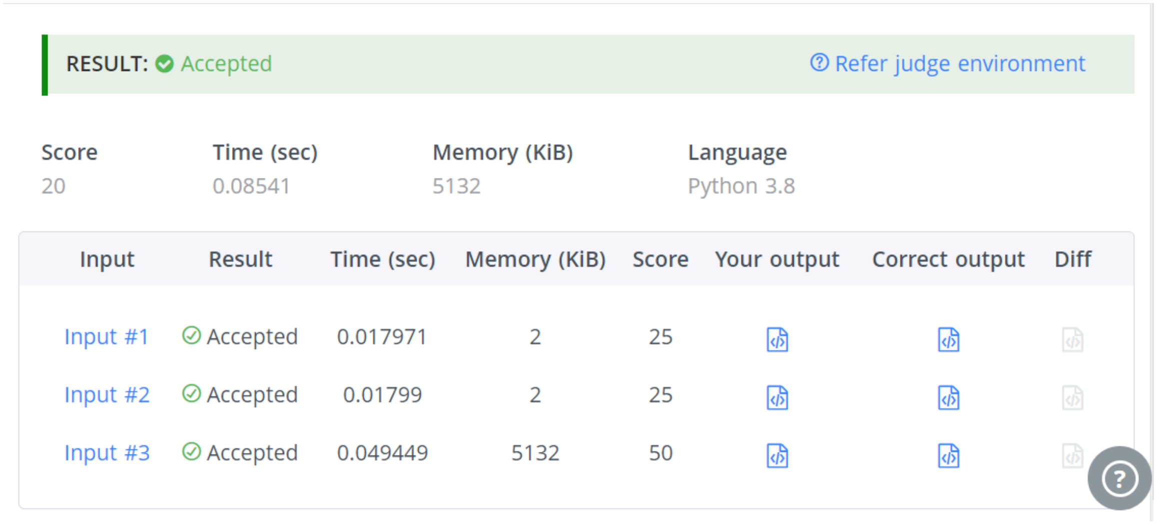
```
sda
sadasda
```

Output

```
1
```

Expected Correct Output

```
1
```



RESULT: Accepted

Refer judge environment

Score	Time (sec)	Memory (KiB)	Language
20	0.08541	5132	Python 3.8

Input	Result	Time (sec)	Memory (KiB)	Score	Your output	Correct output	Diff
Input #1	Accepted	0.017971	2	25			
Input #2	Accepted	0.01799	2	25			
Input #3	Accepted	0.049449	5132	50			

Outcomes: Understand the Graphs, related algorithms, efficient implementation of those algorithms and applications

---

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

The KMP Algorithm efficiently searches for patterns within text by "remembering" prior matches, avoiding redundant comparisons. Its use of the LPS table reduces time complexity, making it superior to naive methods. Implementing KMP offers practical insights into algorithm efficiency and its applications, such as database searches and text processing. This experience highlights the significance of employing efficient algorithms for solving complex problems, aligning with our initial objectives.

References:

1. String Searching Tutorials & Notes | Algorithms | HackerEarth
2. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
3. Antti Laaksonen, “Guide to Competitive Programming”,Springer,2018
4. Gayle Laakmann McDowell,” Cracking the Coding Interview”,CareerCup LLC,2015
5. Steven S. Skiena Miguel A. Revilla,”Programming challenges, The Programming Contest Training Manual”, Springer, 2006
6. Antti Laaksonen, “Competitive Programmer’s Handbook”, Hand book, 2018
7. Steven Halim and Felix Halim, “Competitive Programming 3: The Lower Bounds of Programming Contests”, Handbook for ACM ICPC
