NAME: MD. NAYEEM MOLLA

Subject: Algorithm Design and Analysis

Experiment I: String Search

1. Purpose

Apply brute force to solve the problem of string matching, further improve string matching efficiency through time-space trade-offs. Please SUBMIT experimental reports.

2. Main requirements

(1) Write and debug the brute force string matching algorithm and apply it to a specific search application.

(2) Write the string-matching algorithm using input enhancement technology, repeat the above application, and compare and analyze the efficiency of the two.

3. Instrument and equipment

PC-compatible (language-free).

4. Algorithm's principles

✧ **Problem formulation**

Look for the substring's (the value of i) location of the matching pattern in the text.

$$\begin{array}{rl} \textit{Text}\ \ T: & t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1} \\ & \updownarrow \quad \updownarrow \quad \updownarrow \\ \textit{Pattern}\ \ P: & \underline{p_0 \ldots p_j \ldots p_{m-1}} \Rightarrow \end{array}$$

Text: a string of n characters.

Pattern: a string of m characters. (m < n)

✧ **Algorithm strategy**

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is n – m (provided the text positions are indexed from 0 to n − 1). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

✧ **Pseudocode**

ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

✧ Question: Can choose the Horspool algorithm, KMP algorithm or otherwise for string matching algorithms for input enhancement technology.

I want do this with KMP algorithm using language python

Source code:

```
# Python program for KMP Algorithm by MD NAYEEM MOLLA 381
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    #I think create lps[] that will hold the longest prefix suffix
    # values for pattern Thats what i think
    lps = [0]*M
    j = 0 # index for pat[]

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

    i = 0 # index for txt[]
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print ("Found pattern at index " + str(i-j))
            j = lps[j-1]
```

```
        # mismatch after j matches
        elif i < N and pat[j] != txt[i]:

            # they will match anyway
            if j != 0:
                j = lps[j-1]
            else:
                i += 1

def computeLPSArray(pat, M, lps):
    len = 0 # length of the previous longest prefix suffix

    lps[0] # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i]== pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:

            # AAACAAAA and i = 7. The idea is similar
```

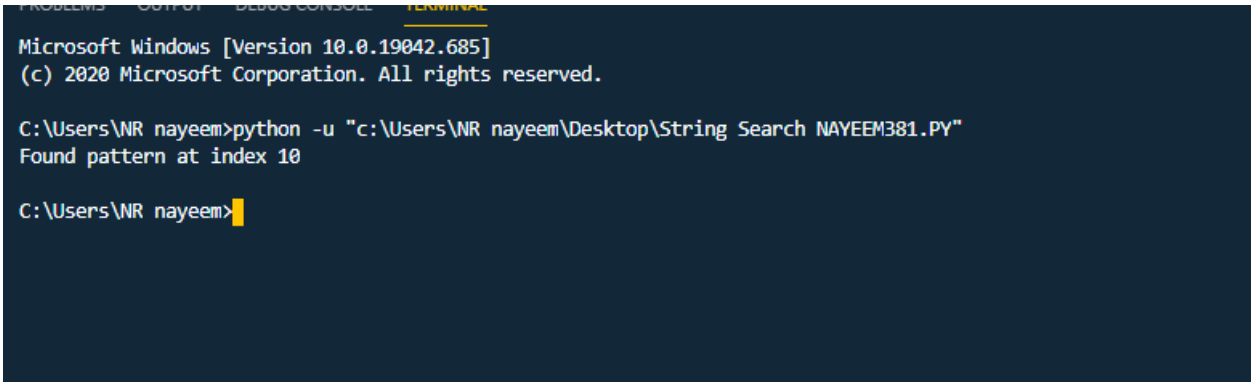
```
            if len != 0:
                len = lps[len-1]

                # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1

txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)
```

INPUT:

```python
# Python program for KMP Algorithm by MD NAYEEM MOLLA 381
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)

    #I think create lps[] that will hold the longest prefix suffix
    # values for pattern Thats what i think
    lps = [0]*M
    j = 0 # index for pat[]

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

    i = 0 # index for txt[]
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1

        if j == M:
            print ("Found pattern at index " + str(i-j))
```

```
            j = lps[j-1]

        # mismatch after j matches
        elif i < N and pat[j] != txt[i]:

            # they will match anyway
            if j != 0:
                j = lps[j-1]
            else:
                i += 1

def computeLPSArray(pat, M, lps):
    len = 0 # length of the previous longest prefix suffix

    lps[0] # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i]== pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:

            # AAACAAAA and i = 7. The idea is similar

            if len != 0:
                len = lps[len-1]

                # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1

txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)
```

OUT PUT


Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Users\NR nayeem>python -u "c:\Users\NR nayeem\Desktop\String Search NAYEEM381.PY"
Found pattern at index 10
C:\Users\NR nayeem>