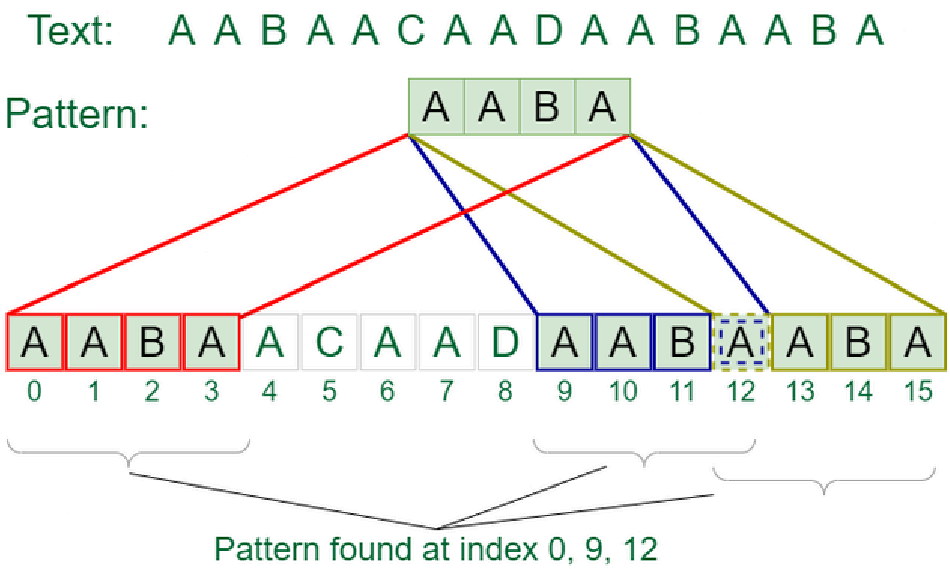
AOA EXPERIMENT 12

Name: Mohammed Arhaan Ali
Div: S11
Roll No: 02

Aim: To study and implement the Knuth-Morris-Pratt Algorithm for String Matching.

Theory:
Given a text txt[0 . . . N-1] and a pattern pat[0 . . . M-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that N > M.
Examples:
Input: txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"
Output: Pattern found at index 10
Input: txt[] = "AABAACAADAABAABA"
pat[] = "AABA"
Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12



The KMP matching algorithm uses the degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n+m).

The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

Matching Overview

txt = "AAAAABAAABA"

pat = "AAAA"

We compare first window of txt with pat

txt = "AAAAABAAABA"

pat = "AAAA" [Initial position]

We find a match. This is same as Naive String Matching.

In the next step, we compare next window of txt with pat.

txt = "AAAAABAAABA"

pat = "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.

Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this,

we pre-process pattern and prepare an integer array lps[] that tells us the count of characters to be skipped

Preprocessing Overview:

- KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as the size of the pattern) which is used to skip characters while matching.
- Name lps indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a 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 subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.
- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the 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].

Note: lps[i] could also be defined as the longest prefix which is also a proper suffix. We need to use it properly in one place to make sure that the whole substring is not considered.

Examples of lps[] construction:

For the pattern “AAAA”, lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”, lps[] is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”, lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “AAACAAAAAC”, lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern “AAABAAA”, lps[] is [0, 1, 2, 0, 1, 2, 3]

Preprocessing Algorithm:

In the preprocessing part,

- We calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index
- We initialize lps[0] and len as 0.
- If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i].
- If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]
- See computeLPSArray() in the below code for details

Illustration of preprocessing (or construction of lps[]):

pat[] = “AAACAAAA”

=> len = 0, i = 0:

- lps[0] is always 0, we move to i = 1

=> len = 0, i = 1:

- Since pat[len] and pat[i] match, do len++,
- store it in lps[i] and do i++.
- Set len = 1, lps[1] = 1, i = 2

=> len = 1, i = 2:

- Since pat[len] and pat[i] match, do len++,
- store it in lps[i] and do i++.
- Set len = 2, lps[2] = 2, i = 3

=> len = 2, i = 3:
- Since pat[len] and pat[i] do not match, and len > 0,
- Set len = lps[len-1] = lps[1] = 1

=> len = 1, i = 3:
- Since pat[len] and pat[i] do not match and len > 0,
- len = lps[len-1] = lps[0] = 0

=> len = 0, i = 3:
- Since pat[len] and pat[i] do not match and len = 0,
- Set lps[3] = 0 and i = 4

=> len = 0, i = 4:
- Since pat[len] and pat[i] match, do len++,
- Store it in lps[i] and do i++.
- Set len = 1, lps[4] = 1, i = 5

=> len = 1, i = 5:
- Since pat[len] and pat[i] match, do len++,
- Store it in lps[i] and do i++.
- Set len = 2, lps[5] = 2, i = 6

=> len = 2, i = 6:
- Since pat[len] and pat[i] match, do len++,
- Store it in lps[i] and do i++.
- len = 3, lps[6] = 3, i = 7

=> len = 3, i = 7:
- Since pat[len] and pat[i] do not match and len > 0,
- Set len = lps[len-1] = lps[2] = 2

=> len = 2, i = 7:
- Since pat[len] and pat[i] match, do len++,
- Store it in lps[i] and do i++.
- len = 3, lps[7] = 3, i = 8

We stop here as we have constructed the whole lps[].

Implementation of KMP algorithm:

Unlike the Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide

the next characters to be matched. The idea is to not match a character that we know will anyway match.
How to use lps[] to decide the next positions (or to know the number of characters to be skipped)?

- We start the comparison of pat[j] with j = 0 with characters of the current window of text.
- We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.
- When we see a mismatch
    - We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increments it only when there is a match).
    - We also know (from the above definition) that lps[j-1] is the count of characters of pat[0…j-1] that are both proper prefix and suffix.
    - From the above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider the above example to understand this.

Below is the illustration of the above algorithm:
Consider txt[] = "AAAAABAAABA", pat[] = "AAAA"
If we follow the above LPS building process then lps[] = {0, 1, 2, 3}
-> i = 0, j = 0: txt[i] and pat[j] match, do i++, j++
-> i = 1, j = 1: txt[i] and pat[j] match, do i++, j++
-> i = 2, j = 2: txt[i] and pat[j] match, do i++, j++
-> i = 3, j = 3: txt[i] and pat[j] match, do i++, j++
-> i = 4, j = 4: Since j = M, print pattern found and reset j, j = lps[j-1] = lps[3] = 3
Here unlike Naive algorithm, we do not match first three
characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.
-> i = 4, j = 3: txt[i] and pat[j] match, do i++, j++
-> i = 5, j = 4: Since j == M, print pattern found and reset j, j = lps[j-1] = lps[3] = 3
Again unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.

-> i = 5, j = 3: txt[i] and pat[j] do NOT match and j > 0, change only j. j = lps[j-1] = lps[2] = 2
-> i = 5, j = 2: txt[i] and pat[j] do NOT match and j > 0, change only j. j = lps[j-1] = lps[1] = 1
-> i = 5, j = 1: txt[i] and pat[j] do NOT match and j > 0, change only j. j = lps[j-1] = lps[0] = 0
-> i = 5, j = 0: txt[i] and pat[j] do NOT match and j is 0, we do i++.
-> i = 6, j = 0: txt[i] and pat[j] match, do i++ and j++
-> i = 7, j = 1: txt[i] and pat[j] match, do i++ and j++
We continue this way till there are sufficient characters in the text to be compared with the characters in the pattern…

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Function to implement the KMP algorithm
void KMP(const char* text, const char* pattern, int m, int n)
{
   // base case 1: pattern is NULL or empty
   if (*pattern == '\0' || n == 0) {
      printf("The pattern occurs with shift 0");
   }

   // base case 2: text is NULL, or text's length is less than that of
pattern's
   if (*text == '\0' || n > m) {
      printf("Pattern not found");
   }

   // next[i] stores the index of the next best partial match
   int next[n + 1];

   for (int i = 0; i < n + 1; i++) {
      next[i] = 0;
```

```
    }

    for (int i = 1; i < n; i++)
    {
        int j = next[i];

        while (j > 0 && pattern[j] != pattern[i]) {
            j = next[j];
        }

        if (j > 0 || pattern[j] == pattern[i]) {
            next[i + 1] = j + 1;
        }
    }

    for (int i = 0, j = 0; i < m; i++)
    {
        if (*(text + i) == *(pattern + j))
        {
            if (++j == n) {
                printf("The pattern occurs with shift %d\n", i - j + 1);
            }
        }
        else if (j > 0)
        {
            j = next[j];
            i--;    // since `i` will be incremented in the next iteration
        }
    }
}

// Program to implement the KMP algorithm in C
int main(void)
{
    char* text = "ABCABAABCABAC";
    char* pattern = "CAB";
```

```
    int n = strlen(text);
    int m = strlen(pattern);

    KMP(text, pattern, n, m);

    return 0;
}
```

Output:

```
PS C:\Users\arhaa\OneDrive\Desktop\AOA> cd "c:\Users\arhaa\One
 }
The pattern occurs with shift 2
The pattern occurs with shift 8
PS C:\Users\arhaa\OneDrive\Desktop\AOA>
```

Conclusion: Thus we have successfully implemented the Knuth-Morris-Pratt string matching algorithm.