Searching for Patterns | Set 2 (KMP Algorithm)

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:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
Pattern found at index 10
```

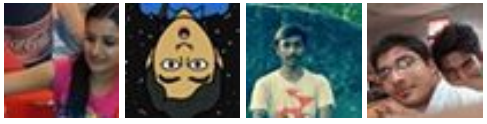
2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

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



We have discussed Naive pattern searching algorithm in the previous post. The worst case complexity of Naive algorithm is O(m(n-m+1)). Time complexity of KMP algorithm is O(n) in worst case.

KMP (Knuth Morris Pratt) Pattern Searching

The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). 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 (since they matched the pattern characters prior to the mismatch). We take advantage of this information to avoid matching the characters that we know will anyway match.

KMP algorithm does some preprocessing over the pattern pat[] and constructs an auxiliary array lps[] of size m (same as size of pattern). Here **name lps indicates longest proper prefix which is also suffix.**. For each sub-pattern pat[0…i] where i = 0 to m-1, lps[i] stores 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].
```

Examples:

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

For the pattern "ABCDE", lps[] is [0, 0, 0, 0, 0]

For the pattern "AAAAA", lps[] is [0, 1, 2, 3, 4]

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

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

Searching Algorithm:

Unlike the Naive algo where we slide the pattern by one, we use a value from lps[] to decide the

Geometric Algorithms

Popular Posts

next sliding position. Let us see how we do that. When we compare pat[j] with txt[i] and see a mismatch, we know that characters pat[0..j-1] match with txt[i-j+1...i-1], and we also know that lps[j-1] characters of pat[0...j-1] are both proper prefix and suffix which means 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. See KMPSearch() in the below code for details.

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.

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

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix values fo
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

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

    int i = 0;  // index for txt[]
    while(i < N)
    {
      if(pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
```

```
            j = lps[j-1];
        }

        // mismatch after j matches
        else if(pat[j] != txt[i])
        {
          // Do not match lps[0..lps[j-1]] characters,
          // they will match anyway
          if(j != 0)
           j = lps[j-1];
          else
           i = i+1;
        }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // lenght of the previous longest prefix suffix
    int i;

    lps[0] = 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++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if( len != 0 )
         {
           // This is tricky. Consider the example AAACAAAA and i = 7.
           len = lps[len-1];

           // Also, note that we do not increment i here
         }
         else // if (len == 0)
         {
           lps[i] = 0;
           i++;
```


Recent Comments

```
            }
        }
    }
}

// Driver program to test above function
int main()
{
    char *txt = "ABABDABACDABABCABAB";
    char *pat = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Related Tpoics:

- Print all possible words from phone digits
- Printing Longest Common Subsequence
- Suffix Array | Set 2 (nLogn Algorithm)
- Rearrange a string so that all same characters become d distance away

- Recursively remove all adjacent duplicates
- Find the first non-repeating character from a stream of characters
- Dynamic Programming | Set 33 (Find if a string is interleaved of two other strings)
- Remove “b” and “ac” from a given string

Writing code in comment? Please use **ideone.com** and share the link here.