Pattern Matching

DATA STRUCTURES & ALGORITHMS

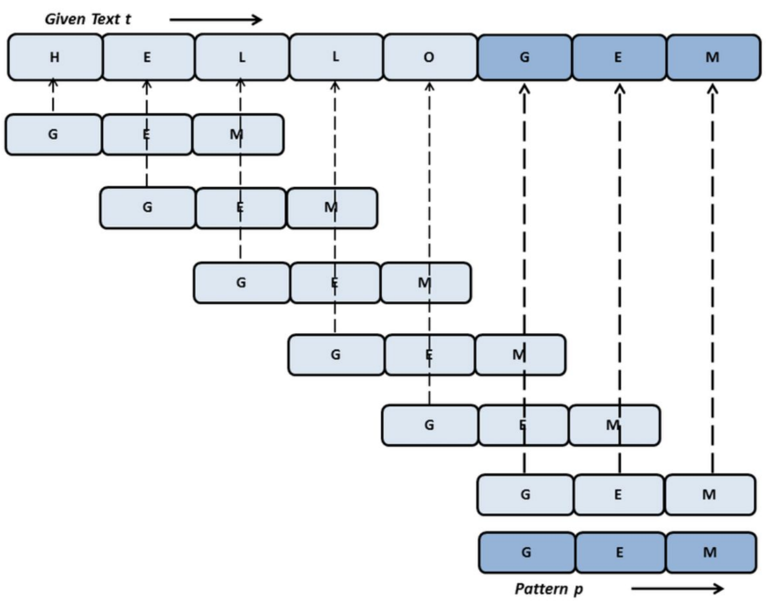
Introduction

- Pattern matching algorithms are computational techniques that locate specific patterns within a dataset, such as strings or sequences, and find their positions or occurrences.
- Applications:
    - Text Search and Retrieval
    - DNA Sequence Analysis
    - Data Mining
    - Speech Recognition
    - Compiler Design

Pattern Matching Algorithms

- Naive Pattern Searching algorithm
- KMP algorithm(Knuth–Morris–Pratt)
- RK algorithm(Rabin Karp)
- Z algorithm

Naive Pattern Searching algorithm




```
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    for i in range(N-M):
        for j in range(M):
            k = j+1
            if(txt[i+j] != pat[j]):
                break
        if(k == M):
            print("Pattern found at index ", i)

txt = "AABAACAADAABAAABAA"
pat = "AABA"
search(pat, txt)
```

KMP Algorithm for Pattern Searching

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

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

LPS examples

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]

LPS calculation steps

keep track of the length of the longest prefix suffix value (use **len** variable for this purpose) for the previous index

- Initialize lps[0] and len as 0.
- If pat[len] and pat[i] match, 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, update len to lps[len-1]

LPS calculation example

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

KMP algorithm

- start the comparison of pat[j] with j = 0 with characters of the current window of text.
- keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching
- When a mismatch occurs:
  1. characters pat[0..j-1] match with txt[i-j…i-1]
  2. lps[j-1] is the count of characters of pat[0…j-1] that are both proper prefix and suffix

SO:

do not need to match these lps[j-1] characters with txt[i-j…i-1] because these characters will anyway match.

KMP algorithm example

Text = “AAAAABAAABA”
Pattern = “AAAA”
=> 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

KMP algorithm example

Text = “AAAAABAAABA”
Pattern = “AAAA”
=> Lps = [0, 1, 2, 3]

- 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
- 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

KMP algorithm example

Text = “AAAAABAAABA”
Pattern = “AAAA”
=> Lps = [0, 1, 2, 3]

- 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++

Rabin-Karp Algorithm

- Unlike Naive string matching algorithm, it does not travel through every character in the initial phase rather it filters the characters that do not match and then performs the comparison
    - Choose a suitable base(b) and a modulus(m)
    - Calculate the initial hash value for the pattern
    - Iterate over each character in the pattern from left to right
    - Calculate hash value as slide window over text
    - Compare hash values
        - If the hash values match, perform a character-by-character comparison to confirm the match

Rabin-Karp Algorithm

Rolling Hash:

$$h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + k[2]b^{L-3} \ldots k[L-1]b^{0}) \bmod m$$

$$h(S_{i+1}) = b(h(S_i) - b^{L-1}S[i]) + S[i+L] \bmod m$$