Please prove that the KMP algorithm has a linear time complexity for finding

all occurrences of pattern P in a string S.

Pseudocode:

prefixFunction(pattern)

{

Lps <- array of size of pattern

Lps[0] <- 0

i <- 0

j <- 1

while j < len(pattern)

{

If pattern[i] = pattern[j]

{

Lps[j] <- i + 1

i <- i+1

j <- j+1

}

Else

{

If i != 0

i <- lps[i-1]

Else

Lps[j] <- 0

j <- j + 1

}

}

}

kmpMatching(T, P)

{

n ← len(T)

m ← len(P)

matchIndices ← empty array

Φ ← prefixFunction(text)

i ← 0

j ← 0

while I < n

{

if T[i] = P[j]

{

i ← i+1

j ← j+1

}

if j = m

{

append i-j to matchIndices

j = Φ [j-1]

}

else if i < n and P[j] != T[i]

{

if j > 0

{

j = Φ [j-1]

}

else

{

i ← i + 1

}

}

}

return matchIndices

}

To understand why the time complexity for KMP algorithm is O (m + n) => O(n), where n >= m, we need to understand the time complexity of the two main sub-parts of the algorithm, which is the time it takes to build the pre-processing array for the given pattern, *p,* and the time it takes to match the traverse the main text, *s,* using the preprocessed array.

Building the preprocessing array will take O(m) time. Looking at the pseudo code, we can see that, no matter what condition, the j pointer advances forward per every iteration until we reached the end of the pattern, so the algorithm is bounded by m, the size of the pattern. And for the number of times the I pointer decrements is at most the number of times the j pointer increments

For matching

-talk about how the i pointer always advances forward and the j pointer moves back to the location (i.e. value at lps[j]) right after the matching prefix since we know that starting from anywhere at the prefix or before is redundant since we would be doing repeated work. We can see, according to the pseudo-code, that the number of times that j decrements in the pattern is tightly bounded by the number of times that i increments in the original text, so given that I is bounded by n, which is the size of the text, and m, which is the size of the pattern, the algorithm would perform at a worst-case runtime of O(m+n) time.

After building the LPS array, we can start finding all occurrences of pattern inside the text. Looking at the pseudo code, we can see that the number of times j decrements is at most the number of times i increments for I < n and j < m. Therefore j is bounded by i and i is bounded by n. So matching will take O(n) time.

Therefore the net total runtime is O(m + n)

Sources:

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

<https://www.youtube.com/watch?v=BXCEFAzhxGY&t=750s> (back to back SWE)

https://www.yumpu.com/en/document/read/22617701/a-correctness-proof-of-the-knuth-morris-pratt-string-matching-algorithm

Please formalize the pseudocode with comments for linear-time Z-value computation with detailed comments.

Please analyze the time complexity of your pseudocode.

Please list left, right, z-value, and which case for each position of the

text S=“aabcaabxaaz”.

The table shows the final values of each variable per kth iteration

|  |  |  |  |
| --- | --- | --- | --- |
| k | Left | Right | Current z array |
| 1 | 1 | 1 | [0,1,0,0,0,0,0,0,0,0,0] |
| 2 | 2 | 1 | [0,1,0,0,0,0,0,0,0,0,0] |
| 3 | 3 | 2 | [0,1,0,0,0,0,0,0,0,0,0] |
| 4 | 4 | 6 | [0,1,0,0,3,0,0,0,0,0,0] |
| 5 | 4 | 6 | [0,1,0,0,3,1,0,0,0,0,0] |
| 6 | 4 | 6 | [0,1,0,0,3,1,0,0,0,0,0] |
| 7 | 7 | 6 | [0,1,0,0,3,1,0,0,0,0,0] |
| 8 | 8 | 9 | [0,1,0,0,3,1,0,0,2,0,0] |
| 9 | 9 | 9 | [0,1,0,0,3,1,0,0,2,1,0] |
| 10 | 10 | 9 | [0,1,0,0,3,1,0,0,2,1,0] |

~~Lets say pattern = aab.~~

~~We know aab appears twice in the string but we need the algorithm to pinpoint those positions for us.~~

~~Lets combine the pattern with the text separated by a character that is non-existent in both strings, so lets choose, for instance, ‘$’~~

~~So the string we will process is a a b $ a a b c a a b x a a z~~

Using the z-algorithm, we will try to find the longest substring starting at the kth position which is also the prefix of the string

Initially:

Z is an array of size n, where n is the size of the string to be processed

Left and right pointers (we’ll call them left and right) start at 0

let k be our iteration variable starting at 1 (because z[0] will be 0) going up to n – 1

for every k we have two choices:

if k > right then we will try keep trying to find the longest substring at position k which is also the prefix of text S and thereby potentially expanding the “Z-box” which is the distance created by the left and right pointers

otherwise there must have been a Z-box previously created by left and right pointers. And in that case, we would check the distance between left and k and use that distance to offset the prefix of S to get the corresponding value in the Z array. And we check if k + that corresponding value is within our right bound. If it is then we can just copy over the value. If not then we have to perform further matching

Again, we start at 1 because the longest substring at index 0 because we have no prefix beforehand to make any comparisons, so starting at index 1, we can look behind and compare

PSEUDO-CODE:

Func process(string):

n ← size of string

z ← array of size n initialized with all zeros

left ← 0

right ← 0

for k ← 1 to n-1

{

if k > right

{

# left and right will now start at k

left ← k

right ← k

while right < n and s[right] = s[right – left]

{

right ← 1

}

z[k] ← right – left

right ← right - 1

}

else

{

# distance between k and left pointer (let’s call it k1)

k1 ← k – left

if z[k] + k <= right

{

z[k] ← z[k1]

}

else

{

# move left pointer up to k

left ← k

# just like before, we look for more matches

while right < n and s[right] = s[right – left]

{

right ← right + 1

}

z[k] ← right – left

right ← right - 1

}

}

}

return z value (array)

# text is the string to be searched and pattern is what string to search for

Func getPatternPositions(text, pattern):

# where ‘$’ is a string that doesn’t exist in both pattern and text

CombinedStr = pattern + ‘$’ + text

returnList = empty list

preprocessed = process(CombinedStr)

let n = size of preprocessed

for every i from 0 to n – 1:

if the value at position i in preprocessed has the same length as that of pattern:

append this value to the returnList

return returnList

Sources:

<https://www.youtube.com/watch?v=CpZh4eF8QBw>(Tuschar roy from youtube)

3. Periodic strings (30%)

For each of the n prefixes of P, we want to know whether the prefix P [1..i] is a

periodic string. That is, for each i, we want to know the largest k > 1 (if there exist

one) such that P [1..i] can be written as αk for some string α. Of course, we also

want to know the period. Give an algorithm to determine this for all n prefixes in

time linear in the length of P.

We can preprocess the the string P with an lps array. This is the first part of the KMP algorithm where at each index from 1 to n-1, where n is the size of string P, we store the value of the length of the longest suffix which is also the prefix of P. The last position (n-1) of the array would be the longest suffix of the the string P which is also a prefix of P. We can leverage that value to see if 1) that there is exists a suffix at that position which is also prefix, and 2) that difference, k, between that length, l, and the length of the string, n, is divisible by n. Below my algorithm will return determine the largest k value for all prefixes of P, and if it doesn’t exist, then I will put -1 as the placeholder.

Pseudocode:

function getPeriodOfAllPrefixes(P)

{

lps ← array of size n initialized with all zeros

i ← 0

j ← 1

while j < n

{

if P[i] = P[j]

{

lps[j] ← i+1

i ← i+1

j ← j+1

}

else

{

if i = 0

{

lps[j] ← 0

j ← j + 1

}

else

{

i ← lps[i-1]

}

}

}

}