String Algorithms

Introduction

In this section, we will primarily walk through two string searching algorithms—**Knuth Morris Pratt algorithm** and **Z-Algorithm**.

Suppose you are given a string **text** of length **n** and a string **pattern** of length **m**. You need to find all the occurrences of the **pattern** in the **text** or report that no such instance exists.

In the above example, the **pattern** string to be searched, that is, **"abaa"** appears at position 3 (0-indexed) in the **text** string **"abcabaabcabac".**

Naive Algorithm

A naive way to implement this pattern searching algorithm is to move from each position in the **text** string and start matching the **pattern** from that position until we encounter a mismatch between the characters of the two strings or say that the current position is a valid one.

In the given picture, The length of the **text** string is 5, and the length of the **pattern** string is 3. For each position from 0 to 3 in the **text** string, we choose it as the starting position and then try to match the next 3 positions with the **pattern**.

Naive pattern matching algorithm

- Run a loop for i from 0 to **n – m**, where **n** and **m** are the lengths of the **text** and the **pattern** string, respectively.

- Run a loop for j from 0 to **m– 1**, try to match the jth character of the **pattern** with (i + j)th character of the **text** string.
- If a mismatch occurs, skip this instance and continue to the next iteration.
- Else output this position as a matching position.

```
function NaivePatternSearch(text, pattern)
    //  iterate for each candidate position
    for i from 0 to text.length – pattern.length

        // boolean variable to check if any mismatch occurs
            match  = True

        for j from 0 to pattern.length – 1
            //  if mismatch make match = False
            if text[i  + j] not equals pattern[j]
                match = False

        //  if no mismatch print this position
            if match == True
                print the occurrence  i
    return
```

Knuth Morris Pratt Algorithm

We first define a prefix function of the string - The prefix function of a string **s** is an array **lps** of length same as that of string **s** such that **lps[i]** stores the length of the maximum prefix that is also the suffix of the substring s[0..i].

For example:

For the pattern “AAABAAA”,

lps[] is [0, 1, 2, 0, 1, 2, 3]

lps[0] is 0 by definition.

The longest prefix that is also the suffix of string s[0..1] which is AA is 1. (Note that we are only considering the proper prefix and suffix).

Similarly, For the whole string AAABAAA it is 3; hence the lps[6] is 3.

Algorithm for computing the LPS array

- We compute the prefix values **lps[i]** in a loop by iterating from 1 to **n – 1.**
- To calculate the current value **lps[i]** we set a variable j denoting the length of best suffix possible till index i – 1. So **j = lps[i – 1].**
- Test if the suffix of length j + 1 is also a prefix by comparing s[j] with s[i]. If they are equal then we assign lps[i] = j + 1 else reduce j = lps[j – 1].
- If we reach j = 0 , we assign lps[i] = 0 and continue to the next iteration .

```
function PrefixArray(s)
        n = s.length;
        //   initialize to all zeroes
        lps = array[n];

    for i from 1 to n – 1
            j = lps[i –1];
            //   update j untill s[i] becomes equal to s[j] or j becomes zero
            while j greater than 0 && s[i] is not equal to s[j]
                j = lps[j – 1];

            //   if extra character matches increase j
            if s[i] equal to s[j]
                j += 1;

            //   update lps[i]
            lps[i] = j;

        //   return the array
        return lps
```

Algorithm for searching the pattern using KMP Algorithm

Consider a new string **S' = pattern + '#' + 'text'** where + denotes the concatenation operator.

Now, the condition for the **pattern** string to appear at a position [i – M + 1... i] in the string text, the **lps[i]** should be equal to **M** for the corresponding position of i in S'.

Note that lps[i] cannot be larger than M because of the '#' character.

- Create S' = pattern + '#' + 'text'
- Compute the lps array of S.'
- For each i from **2*M** to **M + N** check the value of **lps[i].**
- If it is equal to **M,** then we have found an occurrence at the position **i – 2*M** in the string text else,; else continue to the next iteration.

```
function StringSearchKMP(text, pattern)
        //  construct the new string
        S' = pattern + '#' + text

    //  compute its prefix array
        lps = PrefixArray(S')
        N = text.length
        M = pattern.length
```

for i from 2*M to M + N

// longest prefix match is equal to the length of pattern

if lps[i] == M

// print the corresponding position

print the occurrence as i - 2*M

return

Let us understand the above algorithm with the help of the following example:

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

lps[] = {0, 1, 2, 3}

i = 0, j = 0

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 1, j = 1

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++, j++

i = 2, j = 2

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

pattern[i] and pattern[j] match, do i++, j++

i = 3, j = 3

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[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 the Naive algorithm, we do not match the first three characters of this window. Value of lps[j– 1] (in the above step) gives us an index of the next character to match.

i = 4, j = 3

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[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 an index of the next character to match.

i = 5, j = 3

text[] = "AAAAAZAAAZA"

pattern[] =   "AAAA"

text[i] and pattern[j] do NOT match and j > 0, change only j

j = lps[j– 1] = lps[2] = 2

i = 5, j = 2

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] do NOT match and j > 0, change only j

j = lps[j– 1] = lps[1] = 1

i = 5, j = 1

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] do NOT match and j > 0, change only j

j = lps[j– 1] = lps[0] = 0

i = 5, j = 0

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] do NOT match and j is 0, we do i++.

i = 6, j = 0

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++ and j++

i = 7, j = 1

text[] = "AAAAAZAAAZA"

pattern[] = "AAAA"

text[i] and pattern[j] match, do i++ and j++// the algorithm continues in the same fashion

Z-Algorithm

Link : [https://www.codingninjas.com/codestudio/problems/z-algorithm_1112619]

Problem Statement: You are given a string S of length N and a string P of length M. Your task is to find the number of occurrences of string P in the string S.

Note:

The string only consists of lowercase English alphabets.

Example:

Sample Input:

1

5 2

ababa

ab

Sample Output:

2

Explanation:

String **ab** occurs two times in the string **"ababa"**.

The first occurrence is from position 1 to position 2, and the second occurrence is from position 4 to position 5.

Approach 1: Brute Force Approach

A basic approach is to check all substrings of string **S** and see if any of the substrings is equal to **P**.

Steps:

1. Initialize an integer variable **count** = 0.
2. Start by running an outer loop (loop variable *i*) from 0 to **N – M**.
3. Then run a nested loop(loop variable *j*) from 0 to **M**.
4. For index *i*, check if **S[*i* + *j*] != P[*j*]**; if this condition is true, we simply break the inner loop and move to the next *i*. We do this to check every substring of length **M** if it is equal to **P** or not.
5. Later on, we check if *j* is equal to **M** or not; if this condition is true, we increment our **count** by one because we have found a valid substring of length **M** which is equal to string **P**.
6. Finally, we return the **count,** which is our final answer.

Time Complexity: O(N * M), where **N** and **M** are the lengths of String **S** and **P,** respectively.

Since for every substring, we are checking if it is equal to the string **P** or not; therefore, the time complexity is of **O(N * M).**

Space Complexity: O(1), We are not using any auxiliary space.

Approach 2: Z-Algorithm Approach

In this approach, we maintain a **Z array**. The **Z array** for any string **S** of length **N** is an array of size **N** where the *i*th element is equal to the greatest number of characters starting from the position *i* that coincide with the first characters of **S.**

The idea here is to concatenate both **S** and **P** and make a string **P#S**, where **'#'** is a special character that we are using. Now, for this concatenated string, we make a **Z** array. In this **Z** array, if the value at any index **(z[i])** is equal to **M**, we increment our **answer** (initially = 0) by 1 because this indicates that **P** is present at that index.

Steps:

1. Start by making a concatenated string **C**, where **C = P + # + S** and initialize an integer variable **count = 0.**
2. Make an array (**Z** array) of size **K**, where **K** is the length of string **C**.
3. Now initialize two integer values **L = 0 and R = 0.** We will be using these variables to create an interval to check if the string inside this interval matches **P** or not.
4. Now run a loop (loop variable *i*) from 1 till **K** and check if **i > R** or not.
    1. If **i > R**. This means that there is no substring that is starting before *i* and ending after *i*. So we reset the values of **L** and **R** and calculate this new interval of **[L, R]** by using the Z array method where we use a while loop. We simply check if **C[R – L] = C[R]** and **R < K** (to check if we are still inside the string **C**), and till the time both of these conditions are met, we simply increment our **R** by 1 and finally obtain the Z array element(**z[i]**) by using **z[i] = R – L** and then we decrease **R** by 1.
    2. If **i <= R**. Start by making another integer variable **pos = i – L.**
        1. We first check if **z[pos]** is less than the remaining interval **(R – i + 1)** or not. If it is so then, we simply make **z[i] = z[pos]** because this means that there is no prefix substring that starts at **C[i]** because if it was, then z[pos] would have been larger than the remaining interval **(R – i + 1).**
        2. If **z[pos]** was greater than or equal to the remaining interval **(R – i + 1)**. This condition implies that we can still extend our **[L, R]** interval. To do so, we set **L = i** . Now we start with **R** and manually check as we did in the case of **i > R** where we calculated **z[i] = R – L,** and then we decrease **R** by 1.

5. Since our **Z** array is created, we just traverse our array once and check if the current value of the array is equal to **M** or not. If it is equal to **M**, we increment the **count** by one.
6. Finally, we return **count**, which is our final answer.

Let us understand the above algorithm with the help of the following example :

For example- String S = abba and String P = ab

Here **N = 4 and M = 2**

1. We start by making a string **C = ab + “#” + abba = ab#abba**
2. Make a Z array, whose size is **K**, where **K** is the length of the string **C** (here, **K** = 7) and initialise it with zero.
3. Make two integer variables **L** = 0 and **R** = 0.
4. Run a loop from 1 till **K** (loop variable *i*) and start by checking if *i* less **R** or not.
5. Since our **i > R** we reset **L = i** and **R = i**. While **R < K(7)** and **C[R – L]=C[R]** we keep incrementing **R** and after coming out of this while loop we calculate z[1] and since this while loop is never executed because of **C[0] != C[1]**. Hence, **z[1] = R – L = 1 – 1 = 0.**
6. Next for index 2 we see that **i > R** just like index 1. Here also since **C[0] != C[2]** we find **z[2] = R – L = 2 – 2 = 0.**
7. For index 3 still **i < R**. But here we see that **C[0] = C[3]**, hence we increment **R** by 1 and still **C[1] = C[4]**, hence we again increment by 1 and finally since **C[2] != C[5]** we get out of the while loop and calculate **z[3] = R – L= 5 – 3 = 2** and change **R** to 4.
8. For index 4 we see that still **i >= R**. We make another variable **pos = 4 – 3 = 1**. And since **z[1] < R – i + 1, z[4] = z[1] = 0.**
9. Similarly, for the next index, also **z[5]=0.**
10. For the final index **z[6]**. Since our **i>=R**. We make another variable **pos = 6 – 5 = 1**. And since **z[1] < R – i + 1, z[6] = z[1] = 0.**
11. Our final Z array is [0,0,0,2,0,0,1], and in this array, there is only one element equal to **M(2)**, which is our answer.

Time Complexity: O(N + M), where **N** and **M** are the lengths of String **S** and **P,** respectively.

As we are traversing the string **C** only once and the length of C is **N + M**.

Space Complexity: O(K), here **K = N+M** , where **N** and **M** are the lengths of the string **S** and **P** respectively, as we are making an array of length **K**.