Chapter 40: Substring Search

Section 40.1: Introduction To Knuth-Morris-Pratt (KMP) Algorithm

Suppose that we have a *text* and a *pattern*. We need to determine if the pattern exists in the text or not. For example:

```
+-------+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-------+---+---+---+---+---+---+---+---+
|  Text | a | b | c | b | c | g | l | x |
+-------+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+
| Index   | 0 | 1 | 2 | 3 |
+---------+---+---+---+---+
| Pattern | b | c | g | l |
+---------+---+---+---+---+
```

This *pattern* does exist in the *text*. So our substring search should return **3**, the index of the position from which this *pattern* starts. So how does our brute force substring search procedure work?

What we usually do is: we start from the **0th** index of the *text* and the **0th** index of our *pattern and we compare **Text[0]** with **Pattern[0]**. Since they are not a match, we go to the next index of our *text* and we compare **Text[1]** with **Pattern[0]**. Since this is a match, we increment the index of our *pattern* and the index of the *Text* also. We compare **Text[2]** with **Pattern[1]**. They are also a match. Following the same procedure stated before, we now compare **Text[3]** with **Pattern[2]**. As they do not match, we start from the next position where we started finding the match. That is index **2** of the *Text*. We compare **Text[2]** with **Pattern[0]**. They don't match. Then incrementing index of the *Text*, we compare **Text[3]** with **Pattern[0]**. They match. Again **Text[4]** and **Pattern[1]** match, **Text[5]** and **Pattern[2]** match and **Text[6]** and **Pattern[3]** match. Since we've reached the end of our *Pattern*, we now return the index from which our match started, that is **3**. If our *pattern* was: `bcgll`, that means if the *pattern* didn't exist in our *text*, our search should return exception or **-1** or any other predefined value. We can clearly see that, in the worst case, this algorithm would take $O(mn)$ time where **m** is the length of the *Text* and **n** is the length of the *Pattern*. How do we reduce this time complexity? This is where KMP Substring Search Algorithm comes into the picture.

The Knuth-Morris-Pratt String Searching Algorithm or KMP Algorithm searches for occurrences of a "Pattern" within a main "Text" by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The algorithm was conceived in 1970 by Donuld Knuth and Vaughan Pratt and independently by James H. Morris. The trio published it jointly in 1977.

Let's extend our example *Text* and *Pattern* for better understanding:

```
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Index |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  Text |a |b |c |x |a |b |c |d |a |b |x |a |b |c |d |a |b |c |d |a |b |c |y |
+-------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
```

```
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | y |
+---------+---+---+---+---+---+---+---+---+
```

At first, our *Text* and *Pattern* matches till index **2**. **Text[3]** and **Pattern[3]** doesn't match. So our aim is to not go backwards in this *Text*, that is, in case of a mismatch, we don't want our matching to begin again from the position that we started matching with. To achieve that, we'll look for a **suffix** in our *Pattern* right before our mismatch occurred (substring **abc**), which is also a **prefix** of the substring of our *Pattern*. For our example, since all the characters are unique, there is no suffix, that is the prefix of our matched substring. So what that means is, our next comparison will start from index **0**. Hold on for a bit, you'll understand why we did this. Next, we compare **Text[3]** with **Pattern[0]** and it doesn't match. After that, for *Text* from index **4** to index **9** and for *Pattern* from index **0** to index **5**, we find a match. We find a mismatch in **Text[10]** and **Pattern[6]**. So we take the substring from *Pattern* right before the point where mismatch occurs (substring **abcdabc**), we check for a suffix, that is also a prefix of this substring. We can see here **ab** is both the suffix and prefix of this substring. What that means is, since we've matched until **Text[10]**, the characters right before the mismatch is **ab**. What we can infer from it is that since **ab** is also a prefix of the substring we took, we don't have to check **ab** again and the next check can start from **Text[10]** and **Pattern[2]**. We didn't have to look back to the whole *Text*, we can start directly from where our mismatch occurred. Now we check **Text[10]** and **Pattern[2]**, since it's a mismatch, and the substring before mismatch (**abc**) doesn't contain a suffix which is also a prefix, we check **Text[10]** and **Pattern[0]**, they don't match. After that for *Text* from index **11** to index **17** and for *Pattern* from index **0** to index **6**. We find a mismatch in **Text[18]** and **Pattern[7]**. So again we check the substring before mismatch (substring **abcdabc**) and find **abc** is both the suffix and the prefix. So since we matched till **Pattern[7]**, **abc** must be before **Text[18]**. That means, we don't need to compare until **Text[17]** and our comparison will start from **Text[18]** and **Pattern[3]**. Thus we will find a match and we'll return **15** which is our starting index of the match. This is how our KMP Substring Search works using suffix and prefix information.

Now, how do we efficiently compute if suffix is same as prefix and at what point to start the check if there is a mismatch of character between *Text* and *Pattern*. Let's take a look at an example:

```
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

We'll generate an array containing the required information. Let's call the array **S**. The size of the array will be same as the length of the pattern. Since the first letter of the *Pattern* can't be the suffix of any prefix, we'll put **S[0]** = **0**. We take **i** = **1** and **j** = **0** at first. At each step we compare **Pattern[i]** and **Pattern[j]** and increment **i**. If there is a match we put **S[i]** = **j** + **1** and increment **j**, if there is a mismatch, we check the previous value position of **j** (if available) and set **j** = **S[j-1]** (if **j** is not equal to **0**), we keep doing this until **S[j]** doesn't match with **S[i]** or **j** doesn't become **0**. For the later one, we put **S[i]** = **0**. For our example:

```
            j   i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
```

Pattern[j] and **Pattern[i]** don't match, so we increment **i** and since **j** is **0**, we don't check the previous value and put **Pattern[i]** = **0**. If we keep incrementing **i**, for **i** = **4**, we'll get a match, so we put **S[i]** = **S[4]** = **j** + **1** = **0** + **1** = **1** and

increment **j** and **i**. Our array will look like:

```
               j               i
+---------+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---------+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 |   |   |   |
+---------+---+---+---+---+---+---+---+---+
```

Since **Pattern[1]** and **Pattern[5]** is a match, we put **S[i]** = **S[5]** = **j** + **1** = **1** + **1** = **2**. If we continue, we'll find a mismatch for **j** = **3** and **i** = **7**. Since **j** is not equal to **0**, we put **j** = **S[j-1]**. And we'll compare the characters at **i** and **j** are same or not, since they are same, we'll put **S[i]** = **j** + 1. Our completed array will look like:

```
+---------+---+---+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |
+---------+---+---+---+---+---+---+---+---+
```

This is our required array. Here a nonzero-value of **S[i]** means there is a **S[i]** length suffix same as the prefix in that substring (substring from **0** to **i**) and the next comparison will start from **S[i]** + **1** position of the *Pattern*. Our algorithm to generate the array would look like:

```
Procedure GenerateSuffixArray(Pattern):
i := 1
j := 0
n := Pattern.length
while i is less than n
    if Pattern[i] is equal to Pattern[j]
        S[i] := j + 1
        j := j + 1
        i := i + 1
    else
        if j is not equal to 0
            j := S[j-1]
        else
            S[i] := 0
            i := i + 1
        end if
    end if
end while
```

The time complexity to build this array is `O(n)` and the space complexity is also `O(n)`. To make sure if you have completely understood the algorithm, try to generate an array for pattern `aabaabaa` and check if the result matches with this one.

Now let's do a substring search using the following example:

```
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+
|   Text  | a | b | x | a | b | c | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+---+---+---+---+---+---+

+---------+---+---+---+---+---+---+
|  Index  | 0 | 1 | 2 | 3 | 4 | 5 |
```

```
+---------+---+---+---+---+---+---+
| Pattern | a | b | c | a | b | y |
+---------+---+---+---+---+---+---+
|    S    | 0 | 0 | 0 | 1 | 2 | 0 |
+---------+---+---+---+---+---+---+
```

We have a *Text*, a *Pattern* and a pre-calculated array *S* using our logic defined before. We compare **Text[0]** and **Pattern[0]** and they are same. **Text[1]** and **Pattern[1]** are same. **Text[2]** and **Pattern[2]** are not same. We check the value at the position right before the mismatch. Since **S[1]** is **0**, there is no suffix that is same as the prefix in our substring and our comparison starts at position **S[1]**, which is **0**. So **Pattern[0]** is not same as **Text[2]**, so we move on. **Text[3]** is same as **Pattern[0]** and there is a match till **Text[8]** and **Pattern[5]**. We go one step back in the **S** array and find **2**. So this means there is a prefix of length **2** which is also the suffix of this substring (**abcab)** which is **ab**. That also means that there is an **ab** before **Text[8]**. So we can safely ignore **Pattern[0]** and **Pattern[1]** and start our next comparison from **Pattern[2]** and **Text[8]**. If we continue, we'll find the *Pattern* in the *Text*. Our procedure will look like:

```
Procedure KMP(Text, Pattern)
GenerateSuffixArray(Pattern)
m := Text.Length
n := Pattern.Length
i := 0
j := 0
while i is less than m
    if Pattern[j] is equal to Text[i]
        j := j + 1
        i := i + 1
    if j is equal to n
        Return (j-i)
    else if i < m and Pattern[j] is not equal t Text[i]
        if j is not equal to 0
            j = S[j-1]
        else
            i := i + 1
        end if
    end if
end while
Return -1
```

The time complexity of this algorithm apart from the Suffix Array Calculation is `O(m)`. Since *GenerateSuffixArray* takes `O(n)`, the total time complexity of KMP Algorithm is: `O(m+n)`.

PS: If you want to find multiple occurrences of *Pattern* in the *Text*, instead of returning the value, print it/store it and set `j := S[j-1]`. Also keep a `flag` to track whether you have found any occurrence or not and handle it accordingly.

Section 40.2: Introduction to Rabin-Karp Algorithm

Rabin-Karp Algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin that uses hashing to find any one of a set of pattern strings in a text.

A substring of a string is another string that occurs in. For example, *ver* is a substring of *stackoverflow*. Not to be confused with subsequence because *cover* is a subsequence of the same string. In other words, any subset of consecutive letters in a string is a substring of the given string.

In Rabin-Karp algorithm, we'll generate a hash of our *pattern* that we are looking for & check if the rolling hash of our *text* matches the *pattern* or not. If it doesn't match, we can guarantee that the *pattern* **doesn't exist** in the *text*.

However, if it does match, the *pattern* **can** be present in the *text*. Let's look at an example:

Let's say we have a text: **yeminsajid** and we want to find out if the pattern **nsa** exists in the text. To calculate the hash and rolling hash, we'll need to use a prime number. This can be any prime number. Let's take **prime** = **11** for this example. We'll determine hash value using this formula:

```
(1st letter) X (prime) + (2nd letter) X (prime)¹ + (3rd letter) X (prime)² X + ......
```

We'll denote:

```
a -> 1    g -> 7    m -> 13   s -> 19   y -> 25
b -> 2    h -> 8    n -> 14   t -> 20   z -> 26
c -> 3    i -> 9    o -> 15   u -> 21
d -> 4    j -> 10   p -> 16   v -> 22
e -> 5    k -> 11   q -> 17   w -> 23
f -> 6    l -> 12   r -> 18   x -> 24
```

The hash value of **nsa** will be:

```
14 X 11⁰ + 19 X 11¹ + 1 X 11² = 344
```

Now we find the rolling-hash of our text. If the rolling hash matches with the hash value of our pattern, we'll check if the strings match or not. Since our pattern has **3** letters, we'll take 1st **3** letters **yem** from our text and calculate hash value. We get:

```
25 X 11⁰ + 5 X 11¹ + 13 X 11² = 1653
```

This value doesn't match with our pattern's hash value. So the string doesn't exists here. Now we need to consider the next step. To calculate the hash value of our next string **emi**. We can calculate this using our formula. But that would be rather trivial and cost us more. Instead, we use another technique.

- We subtract the value of the **First Letter of Previous String** from our current hash value. In this case, **y**. We get, `1653 - 25 = 1628`.
- We divide the difference with our **prime**, which is **11** for this example. We get, `1628 / 11 = 148`.
- We add **new letter X (prime)□⁻¹**, where **m** is the length of the pattern, with the quotient, which is **i** = **9**. We get, `148 + 9 X 11² = 1237`.

The new hash value is not equal to our patterns hash value. Moving on, for **n** we get:

```
Previous String: emi
First Letter of Previous String: e(5)
New Letter: n(14)
New String: "min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11² = 1806
```

It doesn't match. After that, for **s**, we get:

```
Previous String: min
First Letter of Previous String: m(13)
New Letter: s(19)
New String: "ins"
1806 - 13 = 1793
1793 / 11 = 163
```

```
163 + 19 X 11² = 2462
```

It doesn't match. Next, for **a**, we get:

```
Previous String: ins
First Letter of Previous String: i(9)
New Letter: a(1)
New String: "nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11² = 344
```

It's a match! Now we compare our pattern with the current string. Since both the strings match, the substring exists in this string. And we return the starting position of our substring.

The pseudo-code will be:

Hash Calculation:

```
Procedure Calculate-Hash(String, Prime, x):
hash := 0                                   // Here x denotes the length to be considered
for m from 1 to x                           // to find the hash value
    hash := hash + (Value of String[m])□⁻¹
end for
Return hash
```

Hash Recalculation:

```
Procedure Recalculate-Hash(String, Curr, Prime, Hash):
Hash := Hash - Value of String[Curr]  //here Curr denotes First Letter of Previous String
Hash := Hash / Prime
m := String.length
New := Curr + m - 1
Hash := Hash + (Value of String[New])□⁻¹
Return Hash
```

String Match:

```
Procedure String-Match(Text, Pattern, m):
for i from m to Pattern-length + m - 1
    if Text[i] is not equal to Pattern[i]
        Return false
    end if
end for
Return true
```

Rabin-Karp:

```
Procedure Rabin-Karp(Text, Pattern, Prime):
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i from 1 to Text.length - m
    if HashValue == CurrValue and String-Match(Text, Pattern, i) is true
        Return i
    end if
    CurrValue := Recalculate-Hash(String, i+1, Prime, CurrValue)
end for
```

```
Return -1
```

If the algorithm doesn't find any match, it simply returns **-1**.

This algorithm is used in detecting plagiarism. Given source material, the algorithm can rapidly search through a paper for instances of sentences from the source material, ignoring details such as case and punctuation. Because of the abundance of the sought strings, single-string searching algorithms are impractical here. Again, **Knuth-Morris-Pratt algorithm** or **Boyer-Moore String Search algorithm** is faster single pattern string searching algorithm, than **Rabin-Karp**. However, it is an algorithm of choice for multiple pattern search. If we want to find any of the large number, say k, fixed length patterns in a text, we can create a simple variant of the Rabin-Karp algorithm.

For text of length **n** and **p** patterns of combined length **m**, its average and best case running time is **O(n+m)** in space **O(p)**, but its worst-case time is **O(nm)**.

Section 40.3: Python Implementation of KMP algorithm

Haystack: The string in which given pattern needs to be searched.
Needle: The pattern to be searched.

Time complexity: Search portion (strstr method) has the complexity O(n) where `n` is the length of haystack but as needle is also pre parsed for building prefix table O(m) is required for building prefix table where `m` is the length of the needle.
Therefore, overall time complexity for KMP is **O(n+m)**
Space complexity: **O(m)** because of prefix table on needle.

Note: Following implementation returns the start position of match in haystack (if there is a match) else returns -1, for edge cases like if needle/haystack is an empty string or needle is not found in haystack.

```
def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimeter = 1
    while(delimeter<n):
        prefix_set.add(needle[:delimeter])
        j = 1
        while(j<delimeter+1):
            if needle[j:delimeter+1] in prefix_set:
                prefix_table[delimeter] = delimeter - j + 1
                break
            j += 1
        delimeter += 1
    return prefix_table

def strstr(haystack, needle):
    # m: denoting the position within S where the prospective match for W begins
    # i: denoting the index of the currently considered character in W.
    haystack_len = len(haystack)
    needle_len = len(needle)
    if (needle_len > haystack_len) or (not haystack_len) or (not needle_len):
        return -1
    prefix_table = get_prefix_table(needle)
    m = i = 0
    while((i<needle_len) and (m<haystack_len)):
        if haystack[m] == needle[i]:
            i += 1
```

```
            m += 1
        else:
            if i != 0:
                i = prefix_table[i-1]
            else:
                m += 1
    if i==needle_len and haystack[m-1] == needle[i-1]:
        return m - needle_len
    else:
        return -1

if __name__ == '__main__':
    needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)
```

Section 40.4: KMP Algorithm in C

Given a text *txt* and a pattern *pat*, the objective of this program will be to print all the occurance of *pat* in *txt*.

Examples:

Input:

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

output:

```
Pattern found at index 10
```

Input:

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

output:

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

C Language Implementation:

```
// C program for implementation of KMP pattern searching
// algorithm
#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 for pattern
    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 (i < N && 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;  // length 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++;
            }
          }
      }
}

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

Output:

```
Found pattern at index 10
```

Reference:

http://www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/