

Knuth Morris Pratt

daa project - analysis

Huzaifa Adnan - 200901092

Mujtaba Abbas - 200901070

Syeda Farwa Rizvi – 200901098

Contents

[Abstract 2](#_Toc106305344)

[Introduction 2](#_Toc106305345)

[Pseudocode For KMP Algorithm 3](#_Toc106305346)

[Explanation: 3](#_Toc106305347)

[Experimental Setup For KMP Algorithm 4](#_Toc106305348)

[Complexity Comparison 4](#_Toc106305349)

[Working Comparison 5](#_Toc106305350)

[Time Complexity For KMP Algorithm 5](#_Toc106305351)

[Best Case Scenario: 5](#_Toc106305352)

[Worst Case Scenario: 6](#_Toc106305353)

[Code 7](#_Toc106305354)

[Conclusion 8](#_Toc106305355)

# Abstract

The goal of this study is to assess the temporal complexity of the Knuth-Morris-Pratt(KMP) string matching method. The challenge of string matching is to find a pattern string inside a bigger string. The KMP method presently has the greatest success in terms of asymptotic time complexity. In this approach, a prefix for the pattern string is generated first, and then the pattern string is found in the bigger string in only linear time using this prefix. Both the prefix calculation and the matcher need two loops, one that traverses the whole text and the other that returns to a specific place in the prefix array. I test these two phases using a huge list of random input strings of varying lengths, and variations of the strings are also controlled for evaluation to imitate various instances. I want to have a detailed examination of the temporal complexity of the KMP method as soon as feasible. The number of fundamental operations is recorded throughout these tests, and then linear testing is carried out, yielding linear regression coefficients and correlation coefficients. Finally, based on the trials and the analysis, it is determined that the linear time complexity is verified.

# Introduction

According to the algorithm's running process, the theoretical and computational analysis may be used to determine the temporal complexity of a specific algorithm. Both techniques count the number of fundamental operations that pay some basic unit of time to decide the temporal complexity. In terms of theoretical analysis, the task can sometimes be as simple as counting the highest set of activities; however, when provisional asset prices are placed, the method of theoretically having to count the max number of operations can become confusing because many apparently doable operations will never arise due to the conditional evaluations, resulting in a significant reduction in the number of total operations. Even yet, if all of the examples are known, theoretical counting may be used to give a precise analysis of the temporal complexity. As a result, the computational estimate is helpful in understanding the time complexity tight analysis. The time complexity of the KMP method is difficult to estimate in this project since there are two apparently loops in each of its two phases, but the time complexity is still proportional to the input size. As a result, it produces a variety of random inputs and computes the number of fundamental operations. It displays the findings and does linear regression after acquiring all of these operations' numbers in relation to the input size. The linear connection in the experiment section is so prominent that no polynomial estimate is required. The idea behind computing time complexity is to count each fundamental process in relation to various inputs. When it comes to creating inputs, two requirements must be met: input strings of a particular length must cover all potential scenarios, and input string lengths must vary from 1 to infinite.

These two requirements are the automatically complete principles for determining the asymptotic time complexity of an algorithm. However, infinite openings with infinite lengths are inconceivable; thus, it replaces the first infinite cases rule with random sampling and the second rule with a very long length. True, it is hard to draw any conclusions about the algorithm's performance just based on my tests, however random sampling plus a large number of trials may still offer an estimate of the ground truth efficiency. The limited experiments, in particular, give analytic indications and work directions. Following the presentation of the tests, a discussion with specific theoretical analysis is offered to support the conclusion that the KMP algorithm's symptomatic time complexity is O in this project (n).

The remainder of the article will cover the KMP algorithm's conception, my technique for testing the two essential parts of the algorithm, and the tests, as well as a commentary.

# Pseudocode For KMP Algorithm

KMP-Algo (T, P) #Text,Pattern

n <- length[T]

m <- length[P]

**1:** pi <- ComputingPrefixTable(P)

**2:** q <- 0

**3：**For i := 0 To n-1

**4：** while q > 0 and P[q] !=T[i] do

**5：** q <- P[q]

**6：** if P[q] = T[i]

Then q ++

**7：** if q = m

Then return i – m + 1

**8：**return –1

## Explanation:

Once using the KMP algo, in the first step, we preprocess the Prefix-Table or Pie-Table of P for calculating the “Failure table,” or it can be termed as the “next array.” In the second step, we initialize q so that the number of characters matches q=0. In the third step, it’s time to begin scanning the string from the left position to the right. Then in the next three steps, the algo begins matching. If the two characters match the equation, the no of matched letters in q, one will be added and continues. Or, if the two don’t match, assign p a new value: q =P [q].

# Experimental Setup For KMP Algorithm

To analyze the efficiency and working of the KMP Algo, we have compared it with the Naïve String-Matching approach. That way, we will get clear results and conclude which is better. Below is the attached picture of the Naïve approach code.

Text

Description automatically generated

This code is written in C language and used for comparison with the KMP Algorithm. In this code, T stands for the string or text, and P stands for the pattern.

## Complexity Comparison

If n is the length of string T & m is the length of string P, then string matching would cost O(mn) run time, which is the worst case.

On the other hand, Knuth-morris-pratt (KMP) matching algorithm runs in O(m+n) time to get all occurrences of pattern P in S.

## Working Comparison

In the KMP algorithm, preprocessing is done in pattern string P & an array of length m is calculated. You find out that in the naive string-matching function, we continually start the search from starting index i. It means that when the search goes wrong in a character before matching the entire P string, we begin from the start of P & we increment the index by 1 in string T to begin the matching again. But in the KMP function, we don’t always begin the search from the starting of pattern P. As we have a pi array. This way, index I doesn’t backtrack like the naïve approach. By analyzing this, KMP is way faster than the naïve approach because naive uses a backtracking approach which enables the Algo to take too much time. On the other hand, KMP uses Prefix Table, and the I index remains at its position and doesn’t backtrack, making it faster to find the pattern in the given string.

# Time Complexity For KMP Algorithm

## Best Case Scenario:

Let us suppose a text and a pattern in order to find the best case time complexity.

Text: ABCDAAAABCDA

Pattern: XYZ ,Text Length: 12 ,Pattern Length: 3

|  |  |  |  |
| --- | --- | --- | --- |
| **Letter** | **X** | **Y** | **Z** |
| **Index** | 0 | 1 | 2 |
| **Value** | 0 | 0 | 0 |

****

As no letter from the pattern is matching the text so that’s why the value of each letter is same that is zero. The comparison will start from zero index. At first, X will compare with A, it is a mismatch so the I will move to next position and j will jump to the next position. After first iteration, X will compare with B, again it is a mismatch so the i will again move to the next position and j will jump to next index. This process will repeat as X,Y,Z doesn’t exist in the pattern. After 10 comparisons, it is confirmed that the pattern doesn’t exist in the text as the pattern is exceeded the text length. So the best case time complexity is calculated below:

T (N) = 0(n-m)+1 ;n is length of text and m is length of pattern

= (12-3)+1

= 10 ;total comparisons

T(N) = O(n) or ;best case

T(N) = 0(n-m)+1 ;best case

## Worst Case Scenario:

Let us suppose a text and a pattern in order to find the worst case time complexity.

Text: ABABABABAB

Pattern: ABAB

Text Length: 10

Pattern Length: 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Letter** | **A** | **B** | **A** | **B** |
| **Index** | 0 | 1 | 2 | 3 |
| **Value** | 0 | 0 | 1 | 2 |

****

When the pattern is found in the text or string then it is a worst case scenario for the KMP Algorithm. As it can be seen, pattern ABAB exists in the given string. Comparison begins at the zero index from I and j loop. At first, A will compare with A, it is a match so the i will move to next position and j also. After first iteration, B will compare with B, again it is a match so the i will again move to the next position and j will jump to next index. In the next iteration, A will compare with A, it is a match so the i will move to next position and j also. Then, B will compare with B, again it is a match so the i will again move to the next position and j will jump to next index. After 4 comparisons the pattern is found. Now The jump is equal to the jth index. The value is 2 so it will move to the 2nd index. Letters will again compare, and the comparisons will be 2. This process repeats until the end of the text or string as the pattern is matching again and again at every step. So there exist a total of 10 comparisons at the end. As we know, the text length is also 10 and it is denoted by n so the worst-case time complexity is expressed below:

T(N) = O(10). ;total comparisons

T(N) = O(n) ;worst case

Note: The best case and worst case are same in KMP Algorithm.

Space Complexity:

S(N) = O(m). ;where m is the pattern length.

Space complexity directly depends upon the length of the pattern given.

# Code For KMP Algorithm

def KMPSearch(pat, txt):

    M = len(pat)

    N = len(txt) #1

    lps = [0]\*M #1

    j = 0 #index for pat[] // #1

    computeLPSArray(pat, M, lps) #1

    i = 0 # index for txt[] #1

    while i < N: #n-1

        if pat[j] == txt[i]: #3(n-1)

            i += 1 #n-1

            j += 1 n-1

        if j == M: n-1

            print ("Found pattern at index", str(i-j))

            j = lps[j-1] #n-1

        #mismatch after j match

        elif i < N and pat[j] != txt[i]: #5n-1

            # Do not match lps[0..lps[j-1]] characters,

            if j != 0: #n-2

                j = lps[j-1] #2

            else:

                i += 1 #1

def computeLPSArray(pat, M, lps):

    len = 0 #1

    lps[0] # lps[0] is always 0 #1

    i = 1 #1

    # loop calculates lps[i] for i = 1 to M-1

    while i < M: #n-1

        if pat[i]== pat[len]: #3n-1

            len += 1 #2

            lps[i] = len #1

            i += 1 #2

        else:

            if len != 0: #2n-1

                len = lps[len-1] #2n-1

            else:

                lps[i] = 0 #2n-1

                i += 1 #2n-1

txt = "ABABABABAB" #1

pat = "ABAB" #1

KMPSearch(pat, txt)

**Output:**

Found pattern at index 3 and after 10 comparisons.

# Conclusion

The KMP algorithm is used in many places, such as to find plagiarism, do digital forensics, check for spelling mistakes, and so on. The simple string matching algorithm will indeed either use a sliding window or a two-pointer approach, which would mean more comparisons. The naive method would take O(mn).

The KMP algorithm is made up of Prefix and Suffix.

LPS Table is used for finding the lengthiest proper prefix which is also a suffix. In this algorithm, we collaborate on the LPS table and the string using two pointers. String[i] and Pattern[j] are compared. Each time through the while loop, one of three things could happen: a. String match, make I and j bigger. b. Strings don't match, but j is greater than 0, so move j to LPS[j - 1] and leave I alone. Then, compare j with i. c. Strings don't match, but j is 0, so increase i and check.

The whole algorithm takes O(m + n) time to run. The only bad thing about the method is that it is hard to understand.