Study On String Matching Algorithm

Rafid Asrar Prottoy
Student Id: 011 132 124
Fatematujjohra
Student Id: 011 132 148
Munshi Allama Rumi
Student Id: 011 132 145

A thesis in the Department of Computer Science and Engineering presented
in partial fulfillment of the requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

United International University
Dhaka, Bangladesh
MonthYear

Declaration

We, Rafid Asrar Prottoy, Fatematujjohra, Munshi Allama Rumi, declare that this thesis titled, Thesis Title and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a BSc degree at United International University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at United International University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

…….

Rafid Asrar Prottoy

Student ID:011 132 124

Dept. Of CSE

…….

Fatematujjohra,011 132 148,Dept. Of CSE

[Name, Student ID and Department of the student]

…….

Munshi Allama Rumi,011 132 145,Dept. Of CSE

[Name, Student ID and Department of the student]

Certificate

I do hereby declare that the research works embodied in this thesis entitled "**Study on string matching algorithm**" is the outcome of an original work carried out by Rafid Asrar Prottoy, Fatematujjohra and Munshi Allama Rumi under my supervision.

I further certify that the dissertation meets the requirements and the standard for the degree of BSc in Computer Science and Engineering.

Dr. Mohammad Nurul Huda
B.Sc, M.Sc Engg.(CSE,BUET)
PhD(Electronics & Information Engg.,TUT,Japan)
Professor &MSCSE Coordinator
Dept. of CSE
United International University
[Name and designation of Supervisor]

Abstract

The concept of string matching algorithms is playing a vital role in finding a pattern into a very large body of text in this digital world. Its application covers a wide range and some of them are highly needed in computer applications. Almost every section even in bioinformatics, detecting plagiarism, information security, pattern recognition string matching algorithms are playing their role effectively. The history of string matching algorithm is reached because its necessary in day to day life is beyond imagination. In this paper, we will talk about some well-known string matching algorithm, the importance of them, problems of them and we will present our string matching algorithm with better runtime and moderated technique. We will cover the run time of string matching algorithms and how they work and most importantly why we need a better working string matching algorithm.

Acknowledgments

First and Foremost I would like to thank Dr. Mohammad Nurul Huda sir for being my supervisor and having the time and patience to explain just what I needed, and just when I needed it. Without his support and proper guidance, this work would not have been possible. I have been benefited immeasurably from his scientific insight and scholarly attitude. His scientific motivation inscribed me to do an extensive study on this thesis. I want to thank all my department teachers for their support. Without their support, it could be difficult to achieve this success. Lastly, I would like to thank my parents and thesis group members and I am indebted to them for all their immense love and support.

Table of Contents

List Of Figures

List Of Figures

List Of Figures

List Of Tables

Chapter 1

Introduction

In this digital era, searching is a common scenario that we all do for finding something in the digital world. But finding the given string pattern from a very big text or an enormous text is an ancient and fundamental problem. That's why string matching plays a crucial role in computer science and it provides a given pattern in a comparatively large text. String matching is also applicable in text processing, information retrieval, information analysis and music retrieval in bioinformatics. The string matching is most distend problem because of its necessity in the digital world. Variation of string matching is many and their behaviors are also not alike. One of the most intrinsic of them is to find a pattern to match exactly identical pattern which is embedded in the large text. This type of algorithm is used to find a definite pattern in a large defined text. However, this type of searching algorithm may not adequate in all applications if the pattern string or defined text may contain typographical errors. The exact identical pattern is needed in the field of bioinformatics where a small variation is important such in DNA or protein sequences.

Dealing with a string matching algorithm the first and foremost duty for everyone is to know what is string matching algorithms? Conventionally it is named after string searching algorithms which are defined by an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. Evolving with time these illustrated algorithms have been overly simplified by painstaking efforts of numerous scientists. Only the definition of string algorithm is not sufficient to get the clear picture of how does this prominent algorithm work. To understand the transparent idea about this one needs to understand its practical approach. Like an example the most basic case of string searching involves one large string, sometimes called the "haystack", and one string which is very short sometimes called the "needle". The goal is to find one or more occurrences of the "needle" within the "haystack". For example, one might search for "to" within:

“Some books are to be tasted, others to be swallowed, and some few to be chewed and digested.” In this example, one might demand the first pattern search, which is in the fourth word in the text other might demand all the occurrences of the pattern in the text which is three and someone might request to find or search the last occurrence in the large text, which is the fourteenth word in the defined text.

There are several types of string searching algorithms- Naïve string search algorithm, Rabin–Karp string search algorithm, Boyer–Moore string search algorithm, Knuth–Morris–Pratt algorithm, Finite automata algorithm, Z algorithm (Linear time pattern searching Algorithm) etc. [1]. Among those Rabin–Karp string search algorithm, Boyer–Moore string search algorithm, Knuth–Morris–Pratt algorithm are widely used.

For Rabin–Karp string search algorithm, it is known as a string-matching algorithm performs well in practice and that also extrapolate to other algorithms for related problems, such as two-dimensional pattern matching Unlike Naïve string search algorithm, Rabin-Karp algorithm makes a hash value for the pattern that is given by user and also the large text that is predefined. Shifter will shift the counter value and try to match with the hash value of pattern and the hash value of the text. If match then tries to match character by character. The worst-case running time of the Rabin-Karp algorithm is O((n - m + 1)m), but in the average case, Rabin-Karp algorithm performance is good enough [2].

For Boyer–Moore string search algorithm it is valid for if the user inputs pattern is comparatively large and the predefined text is also large. Unlike other algorithms, it creates a Bad Match Table which is the main feature of this algorithm. It compares user given pattern with text, starting from a rightmost character in the pattern. If the pattern and text are mismatched then it will move the pattern forward according to the value in the bad match table. The average time complexity of the Boyer–Moore string search algorithm is O(n+m) [3].

Another type of widely known string matching algorithm is Knuth–Morris–Pratt algorithm, which represents a linear-time string-matching algorithm. Their algorithm achieves an (n + m) running time. It is differentiated from other algorithms by avoiding the computation of the

transition function altogether, and it does the pattern matching using just an auxiliary function precomputed from the pattern in time O(m) [4].

Different types of string matching algorithm contain different problem but the most common and unconscionable problem is the runtime or in other word time complexity. Runtime is the heart of the string matching algorithm because if the algorithm takes time then searching pattern into a text will take more time which is not acceptable. So observing this problem many observers are trying to make an algorithm which will take a less run time for searching the pattern into a large text. Our main concern was to make a hybrid algorithm which will do better in the battle of time complexity. We simulate in different ways and try a different technique to make a better working algorithm and finally achieve an algorithm which will give better runtime in the war. So our proposed hybrid algorithm which is called hybrid_BMH is the hybrid form of Boyer–Moore horspool algorithm which was proposed by R. Nigel Horspool in 1980 [5]. These algorithms use in many important fields like web searches, database queries, detecting plagiarism etc. like Boyer–Moore, Boyer–Moore horspool also make a "bad match table" and so do our hybrid one. But the main change is Boyer–Moore horspool algorithm search form the rightmost side and try to make a match and if mismatch then moves according to the "bad match table" and our proposed hybrid algorithm search both sides to match the pattern with the predefined large text. By searching both sides it helps to find the matched once expeditious. On the other hand, it also makes an impact on the runtime as well. The runtime or time complexity of the Boyer–Moore horspool is O(n+m) and the time complexity of our proposed hybrid algorithm named hybrid_BMH is O(n+m)/2. So in the long run (if the given pattern which is given by the user is comparatively large and also the predefined text) our proposed hybrid algorithm gives better performance and runtime.

Chapter 2

Background and Literature Review

The underlying principle behind string matching algorithms is known to all of us - almost everywhere we used this type of algorithm for searching. Many versatile fields like Spell Checkers, Spam Filters / Spam Detection Systems, Intrusion Detection System, Search Engines / Content Searching in Large Databases, Plagiarism Detection, Bioinformatics / DNA Sequencing, Digital Forensics, Information Retrieval string matching algorithms are widely used in a continuous manner. The most basic and underlying concept of string matching is also known as pattern matching, searching user given pattern into a large predefined or user is given text where the pattern is in the text or not. In the digital world, we use this searching algorithm on daily basis but we are not aware that we are using a variation of string matching algorithms.

Data that have to be processed do not corrupt logically into independent records with small identifiable pieces. This type of data is individualized by the fact that it can be written down as a string (a liner typically very long sequence of characters). We have faced string before as they are the very basic data structures in the C language. Strings are the most common thing in word processing system which opens the window for manipulating the ordinary text. Such process text string which might define as the sequences of latter, number, and special characters. This types of object can be very large in volume, for example, this thesis paper contains thousands of charters in it and efficient algorithm plays a vital role in manipulating those thousands of string represented by characters. There are also many types of string and one of them is called binary string which is a fundamental sequence of 0 and 1. Text strings are unlike binary strings because they are holding characters from the English alphabet, for example, A-Z or a-z. So fundamental operation of string matching is pattern matching which deals with given text string construct with length N and user-defined pattern of length M, algorithms duty is to find the occurrence of the pattern length M into the defined text construct with length N. Most of the string matching or pattern matching algorithm can easily extract all the occurrences of the pattern in the defined text

because of the can scan the whole sequence of text sequentially and can find all the point that the pattern is matched with the text [6]. The pattern matching problem can be defined as a searching problem considering the pattern as the key, but string searching algorithm that is already existed in the current digital word does not apply directly because the pattern which is given by user can be long and assemble the text in unknown ways.

The most ancient and costly algorithm brute-force which cost O(nm) at worst case running propagation. The string that arises in many application lead to a running time that is virtually always proportional to O(M+N). Furthermore, it is well suited to good architectural features on the most computer system, so an optimized version provides a standard that is difficult to beat with a clever algorithm.

Chapter 3

Methodology

3.1 - Brute-Force Algorithm

The most familiar method for pattern matching that immediately comes to our mind is just to check, for each possible position in the text at which the pattern could match, whether it does in fact match. The following program searches in this way for the first occurrence of a pattern string p in a text string a:

the pseudo code of Naive String Matching Algorithm:

```
Procedure Naïve_String_Matching(T,P)
  BEGIN
      n=T.length
      m=P.length
      for s=0 to n-m
       if P[1...m]==T[s+1.....s+m]
         print("Pattern occurs with shift %d", s)
  END
End Procedure
```

This program sets up one pointer (i) into the text and another pointer (j) into the pattern. When a matching character is found then pointer i and j will be incremented and continued but whenever

an incongruence occurs during matching process then automatically the pointer j will restore itself at the beginning of the pattern and pointer I will be continuing with the next character of the text and algorithm will move on till end like this. Whenever while loop iteration starts, it sets the value of j equals to 0 and then increment i until a text character matches with the first character of the pattern. When the pattern reaches j= =M then there will be a matching pattern *with* text string at i-M but if the end of the text is reached before the end of the pattern i= =N then we got no match with the text and pattern. If any matching pattern is found in the text then this program will return the index i-M where the matching pattern is found in the text. Almost all text editing application, the loop of the given program iterated and the running time will be approximately proportional to the number of character of the text.

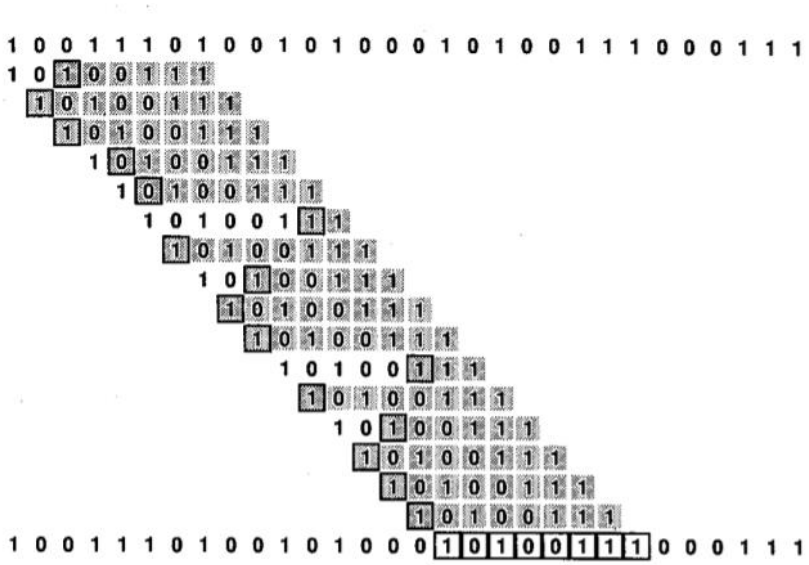


Figure -3.1.1: Brute force string matching algorithm in the binary text. For better examine suppose we are looking at the pattern string which is user defined into the predefined string which is called text.

If we consider a character text and a character pattern by the example below we will be able to know how this Brute force string matching algorithm works with characters.

Let the predefined text is “this is a test” and the user-defined pattern is “test”.

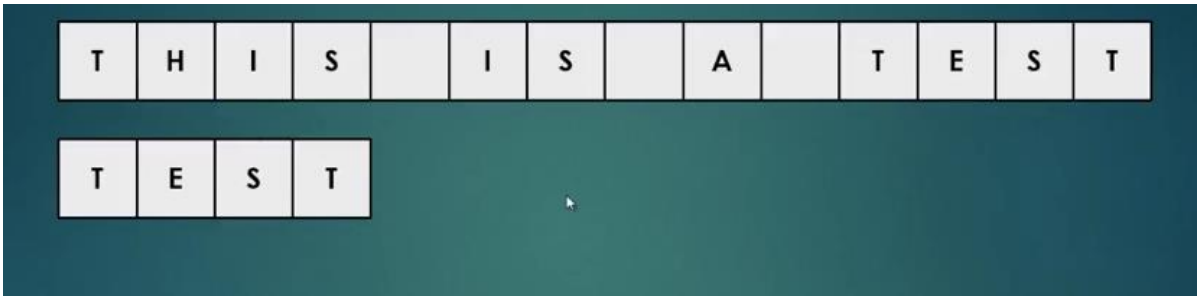


Figure -3.1.2: Let the predefined text is “this is a test” and the user-defined pattern is a test

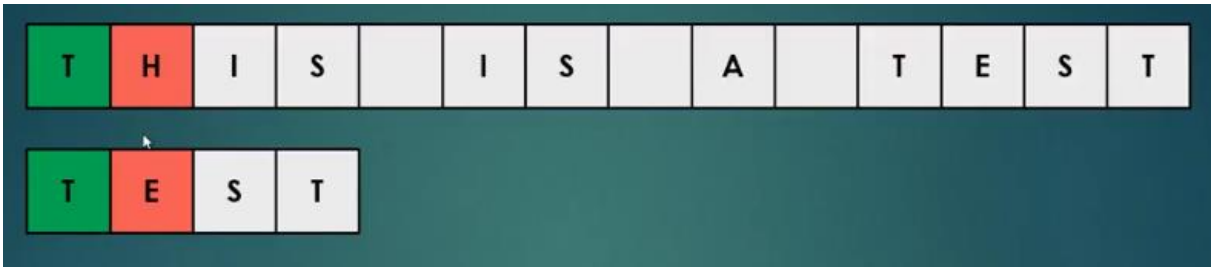


Figure -3.1.3:Here first character “T” is matched with the texts character then we move the counter and after that, the character “H” is not matched with the pattern character “E” so we will move the pattern to the right which is done by changing the pointer value and again we will try to match. If not match then we will continue this process.

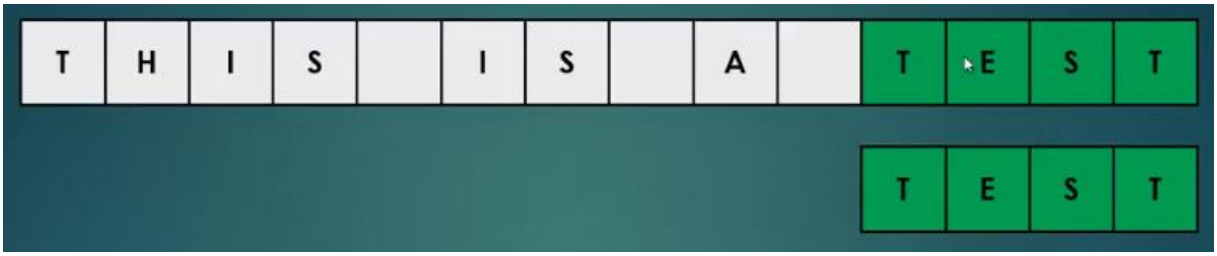


Figure -3.1.4: In this situation, all the pattern are matching with the predefined text. After simulating the algorithm we find the given pattern in the text.

The Brute force string matching algorithm is a very common and old algorithm with the time complexity of O(MN). But it searches character by character that's why it takes more run time and also takes more memory for dealing with every character. Sometimes the last character of the pattern is mismatched with the text, in that case, it shifts once and simulates the loop again. Therefore runtime complexity increases.

3.2 - Rabin–Karp string search algorithm

The Karp–Rabin string searching algorithm or can say Rabin–Karp string searching algorithm is proposed by Richard M. Karp and Michael O. Rabin in early 1982. Unlike other string matching algorithm, it makes a hash value of the pattern as well as predefined large text. As a consequence, it has less runtime complexity than other string matching or searching algorithms. The user will give their desired pattern for search and this Rabin–Karp string searching algorithm makes a hash value of that pattern. After fixing the hash value of the pattern it also introduces a lot of hash values of the predefined large text according to the pattern. Based on this elementary principle, unlike Brute force or other string matching or searching algorithms, it can compare the hash values of the pattern and the text. It takes less time to compare the whole text and pattern as its comparing the hash values instance of character by character check. It uses

comparatively less memory due to hash values take less memory than every character in the pattern. It also defines a hash value for the predefined text according to the pattern. There is no need to keep the whole hash table as we are dealing with the key. To reduce the memory load all we can do is to calculate the hash function for all possible M character section of the predefined text and just check the value is alike the hash value of the pattern or not. If the hash value of the pattern and the text is matched then we can further check character by character so that it takes less time and also less memory load. But finding a hash value for M character is not so much easy as it looks. But Rabin–Karp somehow manage to find an easy way to get rid of the difficulty for the hash function. The method which is used in Rabin–Karp based on calculating the hash function for position i in the predefined text given its value for position i-1, and also follows directly from a mathematical formulation. Assume that we have to translate M characters to numbers by grouping them together in a computer world that we will treat as an integer in future. This translation which will metamorphose the character into an integer will happen base on the based number system, where d is the number of possible characters. The number represented to a[i….i + M-1] is

$$x = a[i]d^{M-1} + a[i+1]d^{M-2} + \cdots + a[i+M-1$$

The pseudo code of Rabin–Karp Matching Algorithm:

Procedure Rabin_karp(T,P)

Begin

n=T.length

m=P.length

hp=hash(P)

ht=hash(T[0……m-1])

for s=0 to n-m

if(hp==ht)

```
    if(P[0.....m-1]==T[0....m-1])
            print("Pattern occurs with shift %d", s)
    if(s<n-m)
    ht=hast(T[S+1……S+m])
    End
End Procedure
```

To make an example of Rabin–Karp string searching algorithm assume that we have a pattern that is given by the user and we have a predefined text and we have to search the pattern into the text using Rabin–Karp string searching algorithm.

Figure -3.2.1: Assume that the predefined text is "GEEKSFORGEEKS" and the user input pattern is "GEEK".

Figure -3.2.2: The text window will shift like this and the length of the window will as same as the length of the pattern size.based on this window length every hash value will be generated for each window and after that, the hash value and every window hash value of the text will be compared.

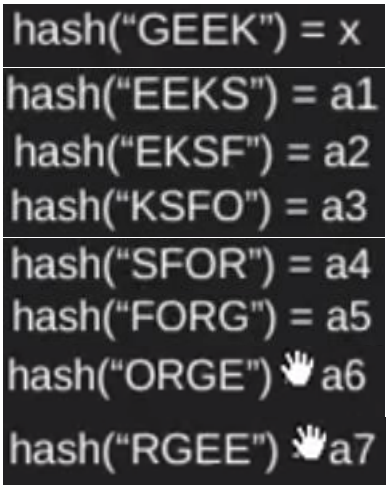


Figure -3.2.3: The hash value of every window in the text will be like this.

The Rabin–Karp algorithm will slide the pattern one by one and try to match the hash value of the pattern with the hash value of current sub-string of the text. If the only hash value of pattern and the hash value of current sub-string of the text is matched then it will try to match character by character of the text and the pattern. The use of hashing will convert the window text to its numeric value which will speed up the equality of the pattern and the sub-string. The algorithm is running with the fact that if two string is equal than their hash values will be also equal. For every string searching algorithms including Rabin–Karp is considering the length of the string N is greater than the length of the pattern M.

Figure -3.2.4: In the first iteration, the hash value of the pattern is matching with the hash value of the sub-string of the text. Because of it is matching now algorithm will try to match character by character and all the characters are matching so algorithm will say the pattern is found in the index 0.

Figure -3.2.5: In the next iteration, the algorithm will slide the window right and again try to match the hash value of the pattern with the hash value of the sub-string of the text. Here they

are not matching.

Figure -3.2.6: After shifting the window the algorithm will again try to match the hash value of the pattern and the hash value of the sub-string of the text and here again not matching so again we will shift the window right.

Figure -3.2.7: Hash values are again not matching so we will again shift the window right.

Figure -3.2.8: Hash values are again not matching so we will again shift the window right.

Figure -3.2.9: Hash values are again not matching so we will again shift the window right.

Figure -3.2.10: Hash values are again not matching so we will again shift the window right.

Figure -3.2.11: In this iteration, the hash value of the pattern is matching with the hash value of the sub-string of the text. Because of it is matching now algorithm will try to match character by character and all the characters are matching so algorithm will say the pattern is found in the index 8. After that, it will shift the window right once.

Figure -3.2.12: Hash values are again not matching and because of it is the last window the algorithm will stop.

Because of Rabin–Karp string matching algorithm is trying to match the hash values there has to be a hash function that will calculate the hash values for the pattern and every the sub-string of the text. Hash at the next shift must be efficiently computable from the current hash value and next character in the text. We are calculating the hash values according to the above equation [8]

$$\boldsymbol{hash(txt[s+1 \ldots\ldots s+m) = d(hash(txt[s \ldots\ldots s+m+1]) - txt[s] * h) + txt[s+m]) mod\ q}$$

here,

hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q

hash(txt[s .. s+m-1]) : Hash value at shift s.

hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: d^(m-1)

The Rabin–Karp string matching algorithm has a very good time complexity which is O(M+N) in the best case. In the worst case when all the characters of the text and the pattern are same have run time complexity of O(MN) [8].

3.3 - Knuth–Morris–Pratt pattern matching algorithm

In modern computer science, the Knuth–Morris–Pratt algorithm or simply KMP algorithm search the pattern occurs within the main text by deeply making the observation that when a mismatch occurs, the word itself put together sufficient information to determine where the next match could begin, by re-examination of past matched characters. This KMP algorithm was first proposed and conceived in 1970 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris and it's an important innovation because it is the first linear time algorithm for string matching timeline. Donald Knuth, Vaughan Pratt, and James H. Morris published it jointly in 1977[12]. In 1969 Matiyasevich[13] discovered a similar algorithm which was coded by a two-dimensional Turing machine when he was studying about a string pattern-matching recognition problem.

The most straightforward algorithm is to look for a character matching at successive values of the index M which is the character index of the pattern. If the index M reaches the end of the text

character then we can say that there is no matching pattern into the text, in this case, the search is said to be “fail”. But if a sub-string of the text is matching with the user given pattern then we can say that we have found a match in the index M. In every case the algorithm will try to match the pattern first character and the text index first character and if match than incrementally try to search rest of the characters of the text and pattern.

But unlike another searching algorithms, this Knuth–Morris–Pratt algorithm first preprocess the pattern and make a failure function. This failure function table leads how much index will shift when a mismatch occurs with the pattern and the text. In the very beginning the algorithm will try to match the pattern character with the text character one by one and if any mismatch occurs then we will shift the pattern right according to failure function value we generate at the very beginning.

pseudo code of Knuth–Morris–Pratt pattern matching algorithm:

```
Procedure KMP(T,P)
    Begin
    n=T.lenght
    m=P.lemght
    π =Computer_Prefix_Function(P)
    q=0                                   //number of characters matched
    for i=1 to n
        while q>0 and P[q+1]!=T[i]
            q= π[q]
        if P[q+1]==T[i]
            q=q+1
        if  q==m
            “Pattern Found”
            q= π[q]
    If q==0
        “Pattern Not Found”
    End
End Procedure
```

In the very beginning, we have the pre-defined text and also the user input pattern that we have to match with the text. From that user-defined pattern first we have to make a failure function table with is looks like figure X.

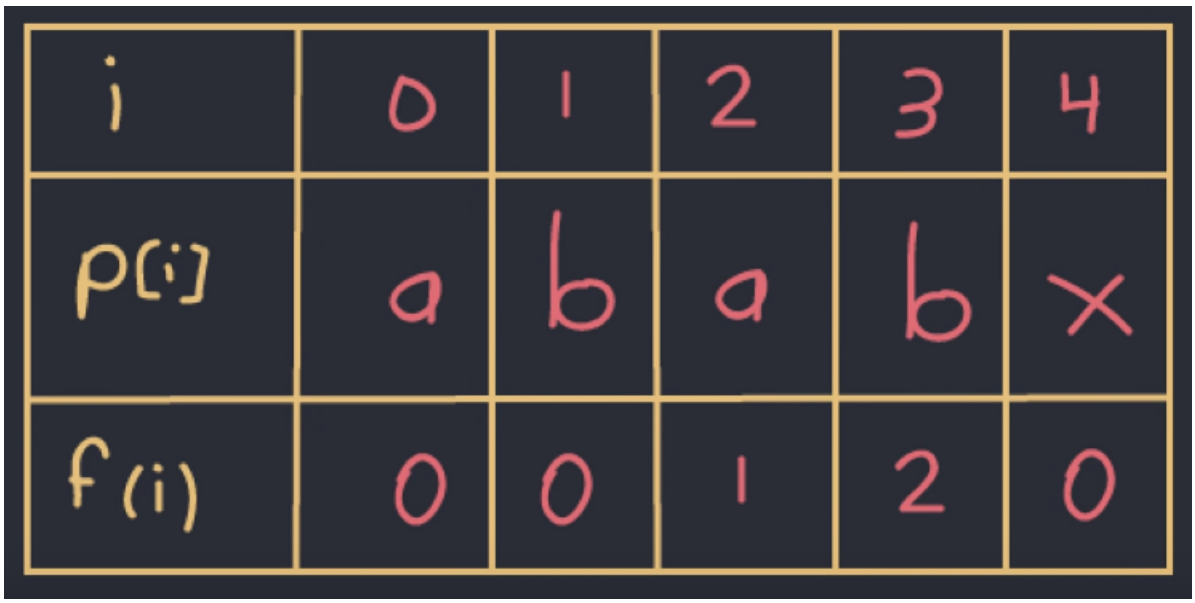


Figure 3.3.1: According to the pattern failure function table.

For an example suppose we have a pre-defined text which is “abtababyababx” and a user-defined text which is “ababx”

In the very beginning, we have to make a failure function table according to the user-defined pattern. In the figure X p[i] is the pattern which is user-defined and f[i] is the failure function number according to p[i]. For first index character “a” is the first character so f[i] is 0. After that in index 1 character “b” is not a prefix so f[i] is 0. In index 2 character “a” is a prefix so f[i] is 1. In index 3 character “b” is a prefix and “ab” is matching so f[i] is 2. In index 4 character “x” is not a prefix so f[i] is 0.

Figure -3.3.2: At the very beginning iteration we are seeing that text character “t” is not matching with the pattern character “a”. the last match shows us that we have to shift the whole pattern into the text index where “t” is located.

Figure -3.3.3: In this iteration, we are seeing that the text character “y” is not matching with the pattern character “x”. So from the failure function, we can see that we have to move the second “a” pattern character into the text character “y” index.

Figure -3.3.4: In this iteration, we are seeing that the text character “y” is not matching with the pattern character “a”. So from the failure function, we can see that we have to move the first “a” pattern character into the text character “y” index.

Figure -3.3.5: In this iteration, we are seeing that the text character “y” is not matching with the pattern character “a”. Because the first thing is mismatched so we have to shift one index.

Figure -3.3.6: Here we can see that all the character of the pattern and text is matching so we can say we have a match found!

In Knuth–Morris–Pratt pattern matching algorithm because of the failure function, we have a very big opportunity to move or shift more than one index and it helps to make the runtime complexity less. In the best case, this has a time complexity of O(N+M). But in the worst case, this algorithm behaves like other and give time complexity of O(NM).

3.4 - Boyer–Moore bad character string search algorithm

The Boyer–Moore bad character string search algorithm is a useful searching algorithm which has a standard benchmark in the field of string searching algorithm [9].This effective string search algorithm is proposed by Robert S. Boyer and J Strother Moore in early 1977 [10].

When other algorithms like naïve and Rabin–Karp are trying to match pattern character and text character from left side and increment but in the Boyer–Moore bad character string search algorithm we are trying to match pattern character and text character from the right side. But shifting the pattern is the same as the other which is the right increment. In Boyer–Moore bad character string search algorithm we learn from character comparisons to skip pointless alignments. Another important component of the Boyer-Moore algorithm is bad character rule and this rule says upon mismatch (we are trying to matching pattern and text and eventually mismatch) we will skip alignment until two things happen, one is if mismatch became a match

and the second one is pattern is moved all the way by mismatching the text character. Hereby trying to match the character of the pattern and the character of the text we are matching right to left and through is matching, if one character mismatch but the character of the text is matching a character in the pattern then we will shift that much index which is coming from that mismatch. If all the character is mismatching then we will shift hole window of the pattern so that we can save time and can skip unnecessary matching. That is the common and effective components that Boyer–Moore bad character algorithm introduce for efficiency.

The pseudo code of Boyer–Moore Matching Algorithm:

```
Procedure BoyerMore(String T, String P)
  Begin
  i = m - 1
  j = m - 1
  repeat
    if p[j] = T[i]
      if j = 0
        return i   // a match
      else
        i = i - 1
        j = j - 1
    else
      i =i + m - min(j , i + last(T[i]))
    until i > n-1
    return –i ;
```

End

End Procedure

In the pseudocode T is the N character of the text and P is the M character of the pattern

Figure -3.4.1: Let the user given pattern is “CCTTTTGC” and the pre-defined text is “GCTTCTGCTACCTTTTGCGCGCGCGCGGAA”.

In the first iteration, the algorithm will try to match the pattern text with the predefined text from the right to left orientation.

Figure -3.4.2: Here pattern character “T” is not matching with the text character “C” but we can see that the text character “C” is matching with the pattern character after two right shift indexes. So the pattern will shift 2 index right and again try to match pattern with the text.

Figure -3.4.3: In this iteration, we are seeing that pattern characters are not matching with the text character so the whole window of the pattern will shift right and again try to match.

Figure -3.4.4: In this iteration, we are seeing all the characters in the pattern are matching with the predefined text. So the algorithm will say the match is found at index 9 and shift hole window left and again try to match the pattern character with the predefined text

This algorithm works well because we are shift according to the bad character rule so we can move more than one index and the runtime will become less. The Boyer–Moore bad character string search algorithm has an effective run time complexity which is O(N+M). Here M is the length of the text and N is the length of the user given pattern which we are searching in the text. But in the worst case, the runtime complexity is O(NM). For example, if the pattern is like "AAAAA" and the text is like "AAAAAAAAAAAAAAA" then it will take more time to execute and the runtime complexity will be O(NM).

3.5 - Boyer–Moore horspool string search algorithm

Boyer–Moore horspool string search algorithm is an algorithm for searching the sub-string in a text which can be user defined or can be pre-defined. This algorithm is proposed and published by Nigel Horspool in 1980 [11]. Like Boyer–Moore Boyer–Moore horspool string search algorithm try to match pattern with the text from the right side but here we are creating a bad match table for shifting the index instant of using bad character rule. Before start matching the pattern with the text Boyer–Moore horspool algorithm makes a bad match table based on the pattern characters and all other characters. Using this bad match table shifter will shift index when a pattern character is not matching with the text character. The problem of Brute-Force Algorithm is we keep considering too many bad options as well, maybe we can eliminate and not count those bad option. That's why Boyer–Moore horspool string search algorithm came to be and it's a very efficient string searching algorithm. This Boyer–Moore horspool string search algorithm needs to preprocess the pattern, but not the whole text! The algorithm runs more comfortably when the length of the pattern increases. A most impressive feature of this algorithm is to match on the tail of the pattern rather than the head.

The performance of the Boyer–Moore horspool string search algorithm is best with long needle strings, when it systematically hits a non-matching character at or near the final byte of the current position in the haystack and the final byte of the needle does not occur elsewhere within the needle. This searching algorithm is useful in web searches, database queries and detecting plagiarism.

The Boyer–Moore horspool string search algorithm works by first constructing a bad match table which will use in the matching. The pattern is compared to the text like other algorithms starting rightmost character in the pattern not left most like other algorithms works. Whenever a mismatch occurs we will shift the pattern left to right corresponding the value of the bad match table which we made at the very beginning. This is the main difference between Boyer–Moore horspool string search algorithm and other string matching algorithms, the Boyer–Moore horspool string search algorithm make a bad match table and move forward the pattern according to the value in the bad match table where other strings matching algorithms are not generating this bad match table(use another method to move forward the pattern index).

the pseudo code of Boyer–Moore Matching Algorithm:

```
Procedure Boyer_Moore()
        Begin
        i<--M-1
        j←M-1
        repeat
          if P[i] = T[j] then
            if j=0 then
              return i {a match found 0}
            else
            i ← i-1
            j ← j -1
          else
          i←i+m-j-1
          i←i+max(j-last(T[i]), match(j))
          j←m-1
        until i > n-1
        return "no match"
        End
End Procedure
```

This pseudo code Boyer–Moore horspool string search algorithm works after making the bad match table according to the pattern character and all the other character that can appear to the text. The bad match table which is needed for this algorithm looks like figure 3.4.1.

Letters	T	E	S	*
Values	1	2	1	4

Figure 3.5.1: Bad match table using in the Boyer–Moore horspool string search algorithm

For an example suppose the pre-defined text is “THIS IS A TEST” and a user tries to search “EXA” in the pre-defined text. Here the user given input “TEST” is the pattern. Now we have to search the pattern into the text according to Boyer–Moore horspool string search algorithm. The iterations are showing below

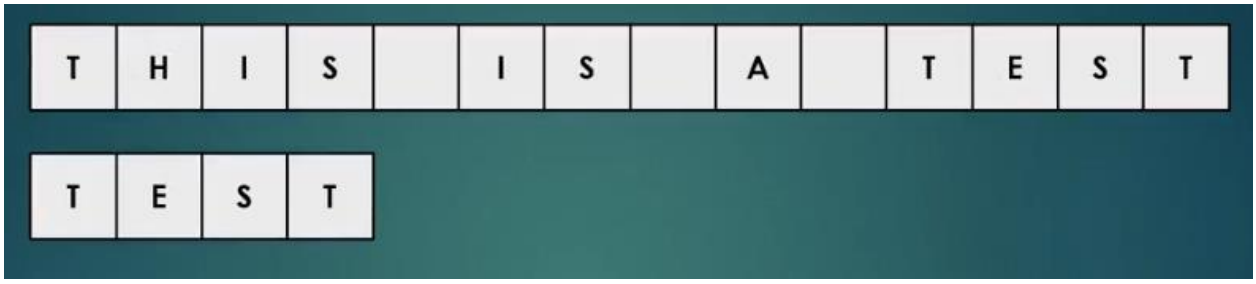


Figure 3.5.2: Here, the pattern is “TEST” and the text is “THIS IS A TEST”.

Figure 3.5.3: At first, we have to make a bad match table.the equation to make a bad match table is max(1,lengthOfPattern-indexOfActualCharacter-1).

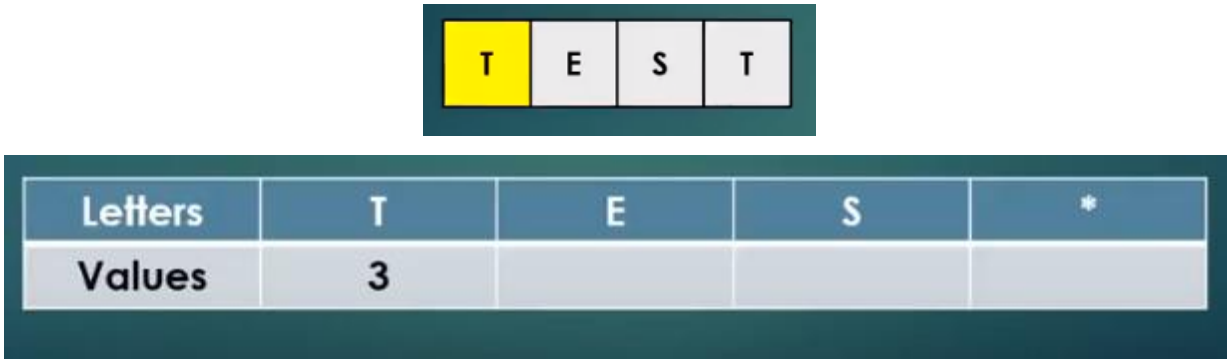


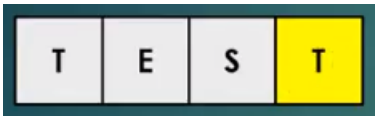
Figure 3.5.4: For the character “T” from the equation, we can see that the length of the pattern is 4 and the index number of “T” is 0 so the result for latter “T” is 3.

Letters	T	E	S	*
Values	3	2		

Figure 3.5.5: For the character “E” from the equation, we can see that the length of the pattern is 4 and the index number of “E” is 1 so result for latter “E” is 2.

Letters	T	E	S	*
Values	3	2	1	

Figure 3.5.6: For the character “S” from the equation, we can see that the length of the pattern is 4 and the index number of “S” is 2 so result for latter “S” is 1.



Letters	T	E	S	*
Values	1	2	1	

Figure 3.5.7: For the character “T” we have already an entry of character “T”. From the equation, we can see that the length of the pattern is 4 and the index number of “T” is 3 so the result for latter “S” is 1 because we are taking the max value (from the equation we can see we are taking the max value).

Letters	T	E	S	*
Values	1	2	1	4

Figure 3.5.8: In the table “*” represents all other latter that can occur in the text. We will find this value in the length of the pattern. The value of LengthOfPattern is “*” character value.

Figure 3.5.9: In the first iteration here we can see pattern character “T” is not matching with the text character “S”. According to bad match table, we can see we have to forward 1 index. So we will slide the pattern 1 index right and again try to match pattern with the text from rightmost.

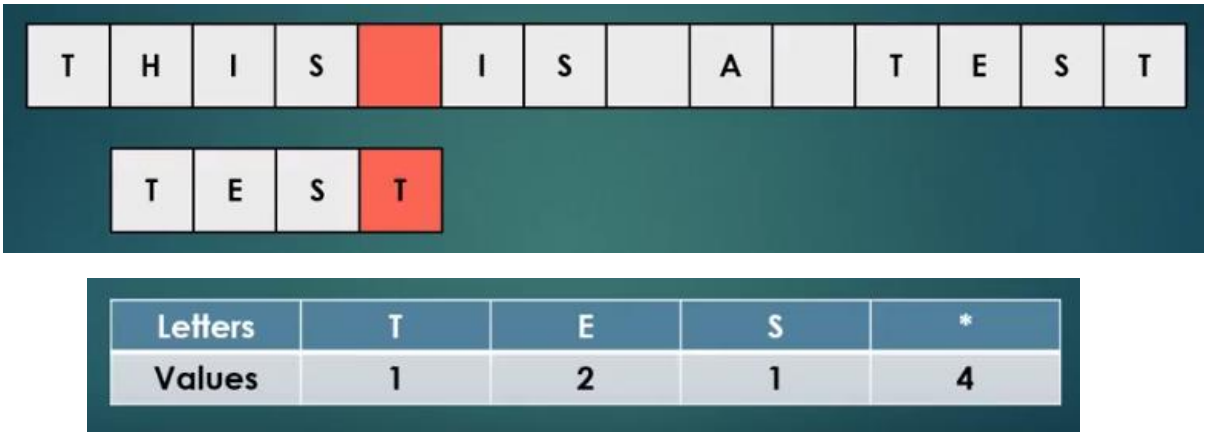


Figure 3.5.10: Here we can see the pattern character “T” is not matching with the text character “ ”. According to bad match table, we can see we have to forward 4 indexes. So we will slide the pattern 4 index right and again try to match pattern with the text from rightmost.

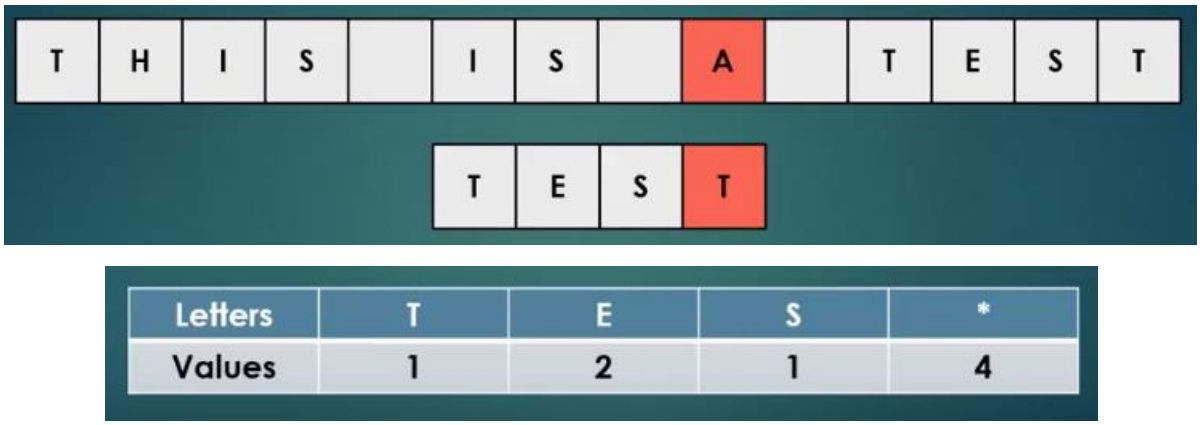


Figure 3.5.11: Here we can see the pattern character “T” is not matching with the text character “A”. According to bad match table, we can see we have to forward 4 indexes. So we will slide the pattern 4 index right and again try to match pattern with the text from rightmost.

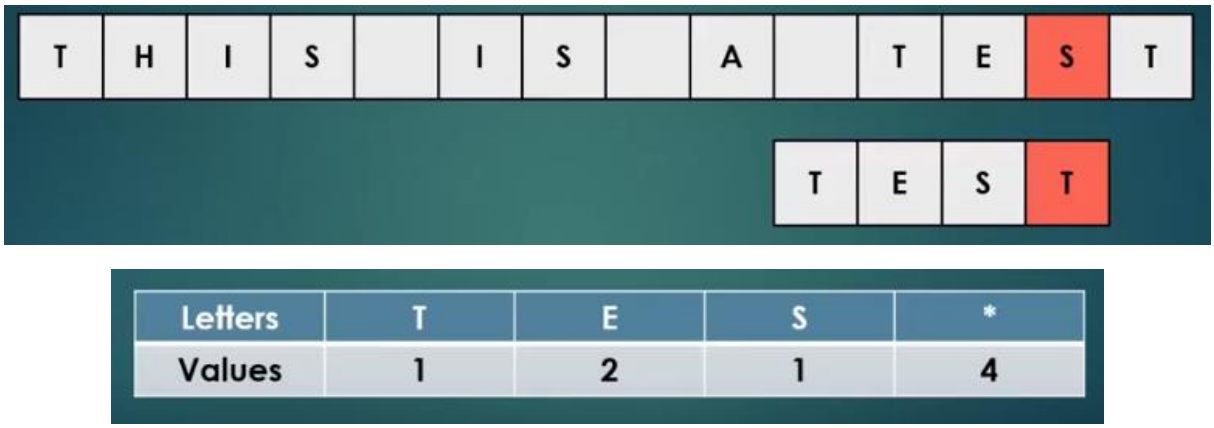


Figure 3.5.12: Here we can see the pattern character “T” is not matching with the text character “S”. According to bad match table, we can see we have to forward 1 index. So we will slide the pattern 1 index right and again try to match pattern with the text from rightmost.

Figure 3.5.13: Here we can see pattern character “T” is matching with the text character “T” and after that, all the characters of the pattern is matching with the characters of the text. So we found a match!

From the algorithm, we can see this algorithm is very efficient because of the bad match table. Using this bad match table we can jump a large amount of index and search time become less. The

Boyer–Moore horspool string search algorithm has a very pleasing runtime which is O(N+M) in average case but in the worst case when the pattern is like “CDD” and the text is like “DDDDDDDCDDDDD” the runtime complexity is same as Brute-Force Algorithm which is O(MN).

Chapter 4

hybrid_BMH string matching algorithm

Almost every string matching algorithm takes a pattern as a user input and the text will be pre-defined or user-defined. Our proposed string matching algorithm which we named hybrid_BMH also takes the pattern from the user and the text will be user-defined or pre-defined. From the user-defined pattern, first, we make a bad match table according to all the characters present in the pattern and also another applicable character which can be present in the text. Our algorithm generates values in the bad match table and from that bad match table which is similar to Boyer–Moore horspool string search algorithm the pattern will shifts right. But unlike all the algorithms if a character from the pattern is not matching or matching with a character of the text from the rightmost side then we also try to match the pattern character with the text character from leftmost side. If pattern character is matching from the rightmost side but pattern character is not matching from leftmost sides then we will shift one index. If pattern character is not matching from the rightmost side and pattern character is also not matching from leftmost sides then we will shift according to the bad match table values. If pattern character is not matching from the rightmost side but pattern character is matching from leftmost sides then we will shift according to the bad match table values. If all the pattern characters and text characters are matching then we found a match.

For this use of the technique, the search of pattern into the text becomes easy and the time is less than other algorithms except for Knuth–Morris–Pratt pattern matching algorithm. Because of checking both sides the search becomes nimble and the iterations become almost half of other string matching or searching algorithms. In this way, our techniques also help to reduce the iteration that affects the runtime. But in the worst case, we have broken ancient run time and introduce a huge less runtime which is $O(NM/2)$.

pseudo code of hybrid_BMH string matching algorithm:

```
Procedure hybrid_BMH()
    Begin
    m=length of text
    n=length of pattern
    hybrid_BMH_search
    while i<m
        while k<n/2
            if pattern[n-1-k]=text[i-k] && pattern[k]=text[i+k-n+1]
                k++
            else if pattern[n-1-k]!=text[i-k]
                miss match from right side
            else if pattern[k]!=text[i+k-n+1]
                miss match from left side
            else
                break
        if k=n/2
            Found
        else
            if miss match found from right
                i=i+bm_table[text[i]]
            else if miss match from left
                i++
    End
End Procedure
```

In the pseudocode, m is the length of the text which is pre-defined and n is the length of the pattern which is user given. In the second while loop, we are checking both side mismatch and take steps depending on which side mismatch happens.

To make an example suppose we have a pre-defined text which is “ABCBBCABC” and a use want to search a pattern “ABC” into this pre-defined text.

First, our algorithm will make a bad match table according to the pattern character and all the possible character that can be present in the text. The bad match table will look like figure:X

A	B	C	*
2	1	3	3

Figure 4.1: Bad match table

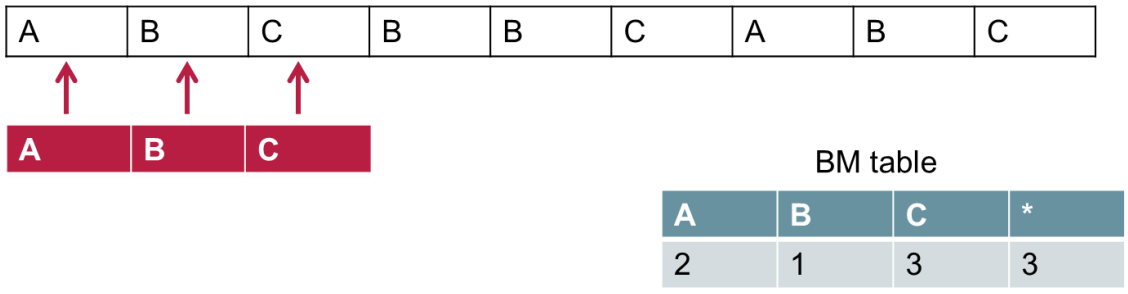
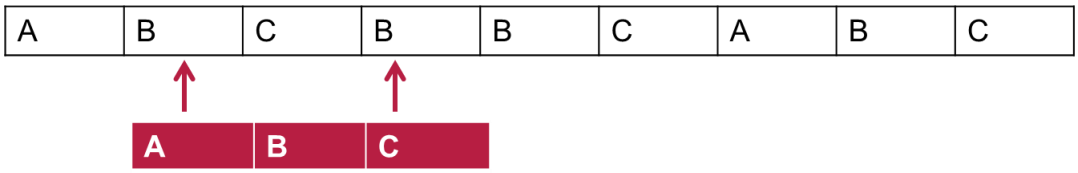


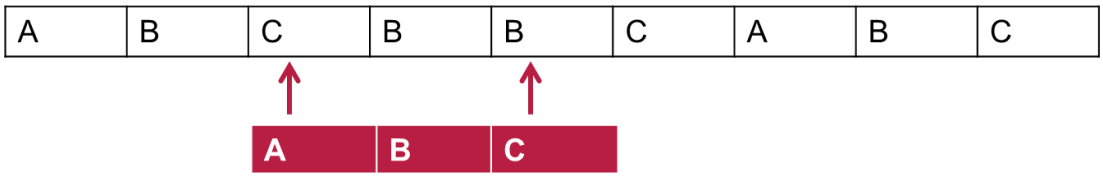
Figure 4.2: In the first iteration, we are seeing that all the characters in the pattern are matching with the characters of the text. So here we have a match found in index 0.



BM table

A	B	C	*
2	1	3	3

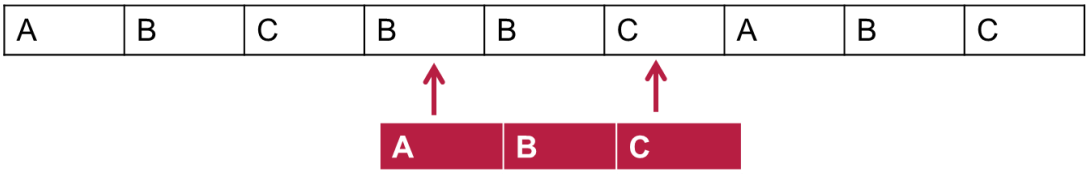
Figure 4.3: In this iteration pattern character “C” is not matching with the text character “B” from the rightmost side and pattern character “A” is not matching with the text character “B” from leftmost side. So we will shift the index according to the bad match table in the right direction.



BM table

A	B	C	*
2	1	3	3

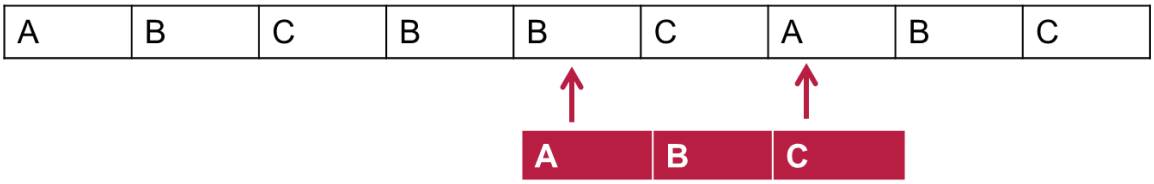
Figure 4.4: In this iteration pattern character “C” is not matching with the text character “B” from the rightmost side and pattern character “A” is not matching with the text character “C” from leftmost side. So we will shift the index according to the bad match table in the right direction.



BM table

A	B	C	*
2	1	3	3

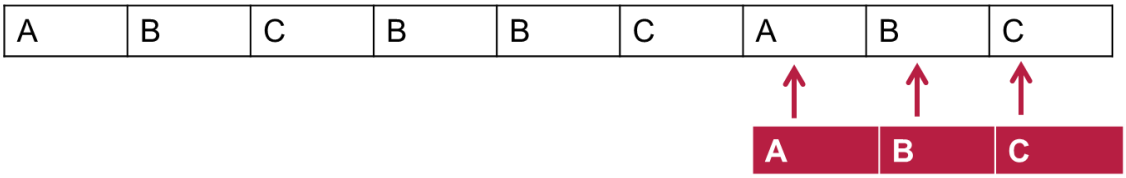
Figure 4.5: In this iteration pattern character “C” is matching with the text character “C” from the rightmost side but pattern character “A” is not matching with the text character “C” from leftmost side. So we will shift the one index in the right direction.



BM table

A	B	C	*
2	1	3	3

Figure 4.6: In this iteration pattern character “C” is not matching with the text character “A” from the rightmost side and pattern character “A” is not matching with the text character “B” from leftmost side. So we will shift the index according to the bad match table in the right direction.



BM table

A	B	C	*
2	1	3	3

Figure 4.7: In this iteration, we are seeing that all the characters in the pattern are matching with the characters of the text. So here we have a match found in index 0.

Our proposed hybrid_BMH string matching algorithm gives a decent runtime which is better than traditional string matching algorithms. In the best which is O((N+M)/2). This achievement is possible for two way checking of the pattern characters into the text characters. Because of this two way checking the iteration become almost half of other string matching algorithm that helps to reduce the run time of the searching pattern into text.

Chapter 5

Result analysis

BMH has a very impressive idea to shift the index according to occur mismatch between pattern and text, we think we can do something with this idea. So we add a condition when checking the mismatch from the right side and the condition is also checked for mismatch from the left side. Here 4 types of case can happen-

- Case-1: Match from the left side and mismatch from the right side.
- Case-2: Mismatch from the left side and match from the right side.
- Case-3: Mismatch from left and right both side.
- Case-4: Match from left and right both side.

If occur case-1 then we shift according to bad match table same as BMH. if occur case-2 then we shift 1 index according to our new idea. If occur case-3 then shift according to bad match table. If the case-4 happens then we go for further checking from both sides.

Runtime Analysis:

Different string matching algorithms gives different runtimes. From them, KMP string matching algorithm takes less time to perform searching. From all string matching algorithms that exist only a few of them don't take any preprocessing time. From the above table, we can see that only BF algorithm don't have the preprocessing time and all other string matching algorithms have preprocessing time. From the table, we can see that our proposed algorithm shows a decent time compared to other string matching algorithm. Our proposed algorithm takes $O(m+|\Sigma|)$ for preprocessing which is similar to BM and BMH algorithms as we use the bad match table concept which is used in both BMH and BM algorithms. In the searching phase, our proposed algorithm did a decent job and takes $O((m+n)/2)$ in the best case. But in the worst case, our proposed one takes $O((nm)/2)$. From the table, we can see that only KMP has a lesser amount of time than our proposed algorithm. All other algorithms except KMP take more time to searching phase than our proposed one.

Algorithms	Preprocessing Phase	Searching Phase	Comparison Order
BF	---	O(mn)	Left to right
FA	O(m+|Σ|)	O(n)	Left to right
RK	O(m)	O(mn)	Left to right
KMP	O(m)	O(n)	Left to right
BM	O(m+|Σ|)	O(mn)	1st right then right to left
BMH	O(m+|Σ|)	O(mn)	1st right then right to left
HBMH	O(m+|Σ|)	Best case: O((m+n)/2) Worst case: O((nm)/2)	1st right then both sides

Table 5.1 Run Time comparisons of various string matching algorithms

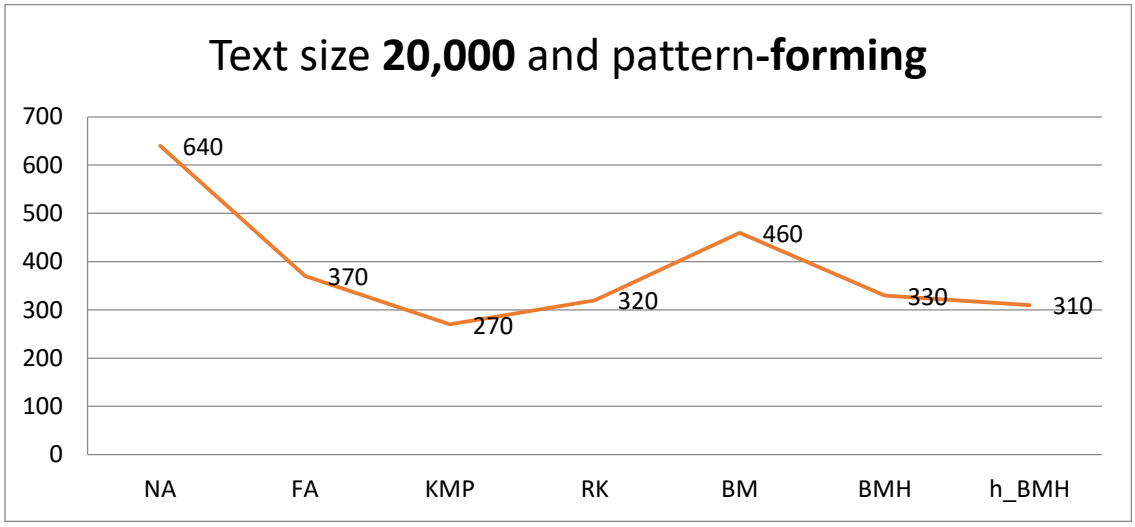


Table 5.1: Table of comparison for 20000 text size

In figure x and figure y, we compared all the algorithms which we describe in the methodology chapter and for 20000 text size in the pre-defined text we search the pattern “forming”. The figure shows us the time of each algorithm take to search the pattern “forming” in a pre-defined text and the text size was 20000 letters. We are seeing that the KMP string matching algorithms

has the most efficient run time for 20000 text size and our proposed h_BMH algorithm is not so far away of this KMP algorithm. Our proposed algorithm shows a little bit extra time then KMP. In the comparison figure our proposed h_BMH has the 2nd position from other 6 algorithms, so we can say that our algorithms work better and produce less time complexity of other 6 algorithms except for KMP, that we talk about in chapter 3.

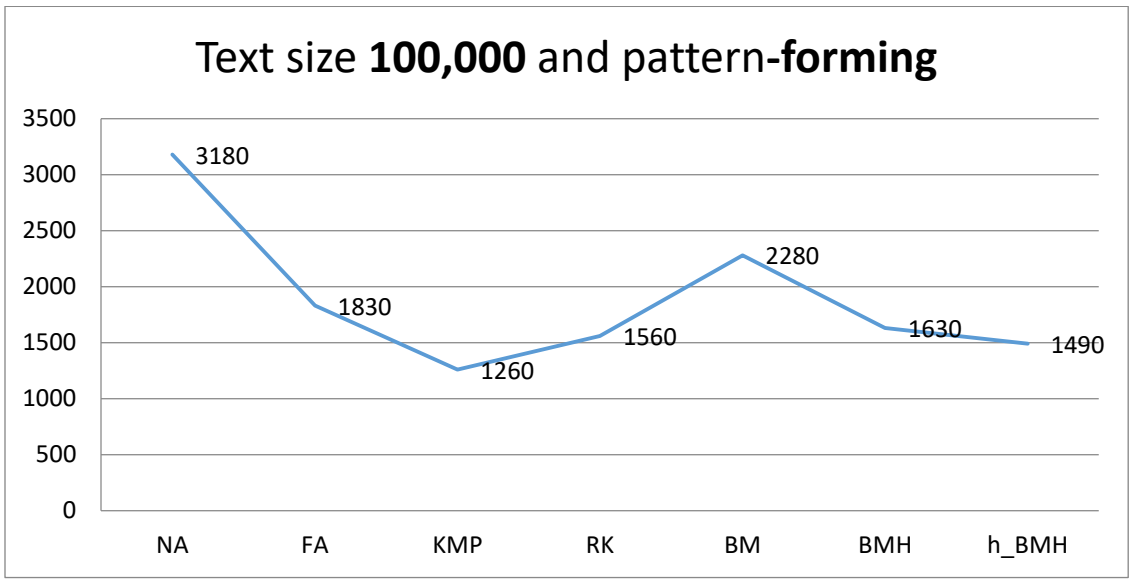


Table 5.2: Table of comparison for 100000 text size

Chapter 6

Conclusions and future work

6.1-Conclusions

String searching is a very commonly used functionality that we use in everyday life. Most of the time we use it without knowing that in the backend we are actually using string matching or searching algorithm. Like whenever we are trying to search a word into a code we normally hit the search button and find the desired word into the code. But we are not a concern that actually we are using a string matching algorithm in the backend.

There are many popular algorithms in the current world that work really well but have an issue of runtimes and that is our main concern too. That's the way we have proposed a new algorithm that we called hybrid_BMH string matching algorithm. This proposed algorithm gives a decent runtime which is almost satisfactory. In the best case scenario, a very first algorithm which is Brute-Force has a runtime of O(MN). There are some other algorithms that work better than Brute-Force and gives a better runtime which is O(N+M). Our proposed algorithm work even better than this runtime. In the best case, our proposed hybrid_BMH string matching algorithm gives a runtime of O((N+M)/2). In the worst case, almost all the string matching algorithms give same time which is O(NM). In this part improvement is possible. From improving the worst case time we work with our algorithm and try to make better performance giver. In the worst case, our proposed algorithm gives a very decent time of O((NM)/2) which is better than existed algorithms. From the graph showed in chapter-5 we can see that in the average case our proposed algorithm works better than other except KMP algorithm. In the graph text size is 100000 and the pattern is "forming". From the runtime, the result is clear to us that our proposed one work better than others and a little bit behind of KMP algorithm.

6.2-Future Works

We think that there is more opportunity to make the runtime better for string matching algorithms. That's why we have some future targets. In the future, we will work on improving the runtime of our algorithm and also try to make our performance better. We want to make a way of shifting the index from left side faster. For this reason, We will try to make a match table which will work to compare left character pattern with text. We think if we can move faster than just one index than we can reduce the iteration and also the time. From this technique, we hope that we can reduce the runtime even better than our algorithm is giving.

Chapter 7

Reference

[1] Robert Sedgewick and Kevin Wayne “Algorithms” Fourth Edition, Addison-Wesley Professional; 4th edition (March 19, 2011); Language: English; ISBN-10: 032157351X; ISBN-13: 978

[2] Richard M. Karp and Michael O. Rabin. “Efficient randomized pattern-matching algorithms” Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[3] Robert S. Boyer and J. Strother Moore “A fast string-searching algorithm” Communications of the ACM, 20(10):762-772, 1977.

[4] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. “Fast pattern matching in strings.” SIAM Journal on Computing, 6(2):323-350, 1977.

[5] R. N. Horspool (1980) "Practical fast searching in strings". Software - Practice & Experience. 10 (6): 501–506

[6] Robert Sedgewick “algorithms in C” Princeton University

[7] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001-09-01) [1990]. "The Rabin–Karp algorithm". Introduction to Algorithms (2nd ed.). Cambridge, Massachusetts: MIT Press. pp. 911–916. ISBN 978-0-262-03293-3.

[8] Karp, Richard M.; Rabin, Michael O. (March 1987). "Efficient randomized pattern-matching algorithms". IBM Journal of Research and Development. 31 (2): 249–260. CiteSeerX 10.1.1.86.9502 Freely accessible. doi:10.1147/rd.312.0249.

[9] Hume; Sunday (November 1991). "Fast String Searching". Software—Practice and Experience. 21 (11): 1221–1248.

[10] Boyer, Robert S.; Moore, J Strother (October 1977). "A Fast String Searching Algorithm". Comm. ACM. New York, NY, USA: Association for Computing Machinery.

20 (10): 762–772. doi:10.1145/359842.359859. ISSN 0001-0782

[11] R. N. Horspool (1980). "Practical fast searching in strings". Software - Practice & Experience. 10 (6): 501–506. CiteSeerX 10.1.1.63.3421 Freely accessible. doi:10.1002/spe.4380100608.

[12] Knuth, Donald; Morris, James H.; Pratt, Vaughan (1977). "Fast pattern matching in strings". SIAM Journal on Computing. 6 (2): 323–350. CiteSeerX 10.1.1.93.8147 Freely accessible. doi:10.1137/0206024.

[13] Cormen, Thomas; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 32.4: The Knuth-Morris-Pratt algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw-Hill. pp. 923–931. ISBN 0-262-03293-7. Zbl 1047.68161.

Chapter 8

Appendix A

Brute Force

```
#include<stdio.h>
#include<string.h>

int main()
{
   int j,k,n,m,flag=0;
   char t[1000],p[100000];
   printf("Enter text: ");
   gets(t);
   printf("\nEnter pattern: ");
   gets(p);
   n=strlen(t);
   m=strlen(p);
   for(int i=0;i<=n-m;i++)
   {
      j=0;
      while(j<m && p[j]==t[j+i])
         {
            j++;
            if(j==m)
            {
               flag=1;
               k=i+1;
               printf("\nPattern found at position: %d\n ",k);
            }
            else
               flag=0;
            }
    }
    if(flag==0)
      printf("\nPattern not found in text \n");
}
```

Finite Automata

```
#include<stdio.h>
#include<string.h>
#include<time.h>
#define NO_OF_CHARS 256
```

```
#define size 20000
int getNextState(char *pat, int M, int state, int x)
{
   if (state < M && x == pat[state])
         return state+1;
   int ns, i;
   for (ns = state; ns > 0; ns--)
   {
      if (pat[ns-1] == x)
      {
         for (i = 0; i < ns-1; i++)
            if (pat[i] != pat[state-ns+1+i])
               break;
         if (i == ns-1)
            return ns;
      }
   }
   return 0;
}
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
   int state, x;
   for (state = 0; state <= M; ++state)
      for (x = 0; x < NO_OF_CHARS; ++x)
         TF[state][x] = getNextState(pat, M, state, x);
}

void search(char *pat, char *txt)
{
   int M = strlen(pat);
   int N = strlen(txt);

   int TF[M+1][NO_OF_CHARS];

   computeTF(pat, M, TF);
   int i, state=0;
   for (i = 0; i < N; i++)
   {
      state = TF[state][txt[i]];
      if (state == M)
         printf ("\n Pattern found at index %d",i-M+1);
   }
}
int main()
{
   clock_t t;
```

```
    char txt[size],pat[size];
    gets(txt);
    gets(pat);
    t=clock();
    search(pat, txt);
    t=clock()-t;
    printf ("\nIt took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
    return 0;
}
```

Rabin-Karp

```
#include<stdio.h>
#include<string.h>
#include<time.h>

#define d 256
#define size 20000

void search(char pat[], char txt[], int q)
{
   int M = strlen(pat);
   int N = strlen(txt);
   int i, j;
   int p = 0;
   int t = 0;
   int h = 1;
   int x = 0;
   for (i = 0; i < M-1; i++)
      h = (h*d)%q;
   for (i = 0; i < M; i++)
   {
      p = (d*p + pat[i])%q;
      t = (d*t + txt[i])%q;
   }
   for (i = 0; i <= N - M; i++)
   {
      if ( p == t )
      {
         for (j = 0; j < M; j++)
         {
            if (txt[i+j] != pat[j])
               break;
         }
         if (j == M)
            {
               printf("Pattern found at index %d \n", i);
               x++;
```

```
                }
            }
            if ( i < N-M )
            {
                t = (d*(t - txt[i]*h) + txt[i+M])%q;
                if (t < 0)
                t = (t + q);
            }

        }
        printf("\n%s Found %d times in main string",pat,x);
    }

    int main()
    {
        clock_t t;
        char txt[size],pat[size];
        gets(txt);
        gets(pat);
        int q = 101;
        t=clock();
        search(pat, txt, q);
        t=clock()-t;
        printf ("It took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
        return 0;
    }
```

Knuth-Morris-Pratt

```
    #include<bits/stdc++.h>
    #include<time.h>
    #define size 20000

    void computeLPSArray(char *pat, int M, int *lps);

    void KMPSearch(char *pat, char *txt)
    {
        int M = strlen(pat);
        int N = strlen(txt);
        int lps[M];
        computeLPSArray(pat, M, lps);

        int i = 0;  // index for txt[]
        int j  = 0;
        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];
        }
        else if (i < N && pat[j] != txt[i])
        {
            if (j != 0)
                j = lps[j-1];
            else
                i = i+1;
        }
    }
}
void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;

    lps[0] = 0;
    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            if (len != 0)
            {
                len = lps[len-1];
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
```

```
}
int main()
{
   clock_t t;
   char txt[size],pat[size];
   gets(txt);
   gets(pat);
   t=clock();
   KMPSearch(pat, txt);
   t=clock()-t;
   printf ("\nIt took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
   return 0;
}
```

Boter-Moore

```
# include <limits.h>
# include <string.h>
# include <stdio.h>
#include<time.h>

#define deferent 20000
# define NO_OF_CHARS 256
int max (int a, int b)
{
   return (a > b)? a: b;
}
void badCharHeuristic( char *str, int size,int badchar[NO_OF_CHARS])
{
   int i;
   for (i = 0; i < NO_OF_CHARS; i++)
       badchar[i] = -1;
   for (i = 0; i < size; i++)
       badchar[(int) str[i]] = i;
}
void search( char *txt,  char *pat)
{
   int m = strlen(pat);
   int n = strlen(txt);

   int badchar[NO_OF_CHARS];
   badCharHeuristic(pat, m, badchar);
   int s = 0;  // s is shift of the pattern with
               // respect to text
   while(s <= (n - m))
   {
      int j = m-1;
      while(j >= 0 && pat[j] == txt[s+j])
```

```
            j--;
        if (j < 0)
        {
            printf("\n pattern found at = %d", s);
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;

        }
        else
            s += max(1, j - badchar[txt[s+j]]);
    }
}
int main()
{
    clock_t t;
    char txt[deferent],pat[deferent];
    gets(txt);
    gets(pat);
    t=clock();
    search(txt, pat);
    t=clock()-t;
    printf ("\nIt took me %d clicks(%fseconds).\n",t,((float)t)/CLOCKS_PER_SEC);
    return 0;
}
```

Boyer-Moore Horspool

```
#include<stdio.h>
#include<string.h>
#include<time.h>

#define d_size 20000
int bm_table[d_size];
void shift_table(char txt[],char pat[])
{
        int i,j,m,n;
        m= strlen(pat);
        n=strlen(txt);
        for (i=0;i<d_size;i++)
        {
                bm_table[i]=m;
        }
        for(j=0;j<m;j++)
        {
                bm_table[pat[j]]=m-1-j;
        }
}
```

```
int BMH (char txt[],char pat[])
{
	int i,k,m,n,ch=0;
	m=strlen(txt);
	n=strlen(pat);
	i=n-1;
	while(i<m)
	{
		k=0;
		while( (k<n) && (pat[n-1-k]==txt[i-k]) )
		{
			k++;
		}
		if (k==n)
		{
			printf("Found at-%d\n",i-n+1);
			ch++;
			i++;
		}
		else
			i+=bm_table[txt[i]];
	}
	return ch;
}
int main()
{
	clock_t t;
	char txt[d_size],pat[d_size];
	gets(txt);
	gets(pat);
	t=clock();
	shift_table(txt,pat);
	int ch = BMH(txt,pat);
	t=clock()-t;
	printf ("It took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
	if (ch==0)
	printf("Pattern not found");
	else
	printf("Found %d times",ch);
	return 0;
}
```

hybrid_BMH

```
	#include<stdio.h>
	#include<string.h>
	#include<time.h>
```

```
#define size 20000

int bm_table[256]={0};
void shift_table(char pat[])
{
   int i,j,m,n;
   m= strlen(pat);
   for (i=0;i<256;i++)
      bm_table[i]=m;
   for(j=0;j<m;j++)
   {
      if (m-1-j==0)
         bm_table[pat[j]]=m;
      else
         bm_table[pat[j]]=m-1-j;
   }
}

int hybrid_BMH (char txt[],char pat[])
{
   int i,k,m,n,z,ch=0,x,y,c_2=1,c_1=1;
   m=strlen(txt);
   n=strlen(pat);
   if(n%2!=0)
      z=(n/2)+1;
   else
      z=n/2;
   i=n-1;
      while(i<m)
   {
      k=0;
      x=0;
      while(k<z)
      {
         if ((pat[n-1-k]==txt[i-k]) &&(pat[x]==txt[i+k-n+1]))
         {
            k++;
            x++;
         }
         else if (pat[n-1-k]!=txt[i-k])
         {
            c_1=0;
            break;
         }
         else if (pat[x]!=txt[i+k-n+1])
         {
```

```
                c_2=0;
                break;
            }
            else
                break;
        }
        if (k==z)
        {
            printf("Found at-%d\n",i-n+1);
            ch++;
            i++;
        }
        else
        {
            if (c_1==0)
            {
                i+=bm_table[txt[i]];
                c_1=1;
            }
            else if (c_2==0)
            {
                i++;
                c_2=1;
            }
        }
    }
    return ch;
}
int main()
{
    clock_t t;
    char txt[size],pat[size];
    gets(txt);
    gets(pat);
    t=clock();
    shift_table(pat);
    int ch = hybrid_BMH(txt,pat);
    t=clock()-t;
    printf ("It took me %d clicks (%f
seconds).\n",t,((float)t)/CLOCKS_PER_SEC);
    if (ch==0)
        printf("Pattern not found");
    else
        printf("Found %d times",ch);
    return 0;
}
```