

Project Report  
On  
*Pattern Matching Algorithm in Genome  
Sequence*

Submitted by:

Jagruthi Patibandla 180001021  
Shravya Ramasahayam 180001052

Computer Science and Engineering  
2<sup>nd</sup> year

Under the Guidance of

*Dr. Kapil Ahuja*



**Department of Computer Science and  
Engineering**

**Indian Institute of Technology Indore**

**Spring 2020**

# **Contents:**

- Introduction
  - Objective
  - Problem Statement
- Algorithm Analysis
- Implementation and Results
  - Brute force Implementation
  - Boyer Moore Implementation
  - KMP Implementation
  - Brute force Results
  - Boyer Moore Results
  - KMP Results
- Complexity Analysis
- Comparison of performances
- Applications of Pattern Matching
- References

## **Introduction:**

Unhealthy symptoms or a specific illness in the body is termed as a disease. The disease may be referred to as disabilities, disorders, syndromes, symptoms. Genes are the basic building blocks of heredity. They get passed from parent to child. They hold DNA, the instructions for making proteins. A genetic disease is any disease that is caused by an abnormality in an individual's genome. Some of the genetic disorders are inherited from the parents.

In contrast, other genetic disorders are caused by mutations in a pre-existing gene or group of genes. DNA sequences of various diseases are stored in databases for easy retrieval and comparison. If it is found that a particular sequence is a cause for a disease, then the trace of the sequence in the DNA and the number of occurrences of the sequence are searched to define the existence and intensity of the disease. As the DNA is an extensive database, molecular biologists are increasingly taking the help of computer science algorithms to find DNA patterns in DNA sequences.

## **Objective:**

Sophisticated pattern matching methods are used in locating DNA sequences, fingerprint assessment, soil patterns reporting, and retinal blood vessel assessment—a well-known application of sequence analysis in bioinformatics. As the DNA is an extensive database, String and Pattern matching algorithms are used to find out a particular sequence in the given DNA. Pattern matching (PM), is a computerised search operation for finding a given pattern within a database. These algorithms are generally used to specify

whether the desired structure is present in the given set of elements or not. This project aims to:

- ✓ To implement these algorithms.
- ✓ To analyse the existing pattern searching algorithms.

## **Problem Statement:**

Given a DNA pattern and DNA, the problem is to find out all the occurrences of the pattern in the DNA.

## **Algorithm Analysis :**

### **1.Brute Force :**

- Consider all the substrings with the length of the given pattern.
- Compare each substring with the given pattern.
- If a match is found, print it.
- We need to check  $n-m$  such substrings each of length  $m$ .

### **2.Boyer - Moore Algorithm (DP):**

#### **Preprocessing:**

- ✓ Construct a bad match table  $p$  of size 26 (alphabets).
- ✓ Initialise the value of every index to the length of pattern (b.size)
- ✓ Update the table as follows (except for the last index) :  
$$p[b[i]] = (\text{length of pattern}) - i - 1$$
- ✓  $p[b[n-1]] = \text{length of the pattern}$

#### **Searching:**

- Initially, the window of text is taken from starting index of text (size same as that of the pattern)

- We start the comparison of pattern with characters of the current window of text both from the right side.
- We keep matching characters of the text window and pattern and keep decrementing the indices till we find a mismatch.
- When we see a mismatch, we use p array to find the next index from where the searching should progress again.
- If we reach the end of the pattern without a mismatch, then print the occurrence and begin the next iteration.

### **Example:**

DNA:     G C A A T G C C T A T G T G A C C  
 Pattern: T A T G T G  
 Index:    0  1  2  3  4  5


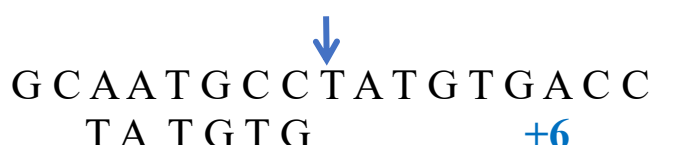
### Preprocessing:

Bad Match Table p:

A	G	T	*
4	6	1	6

(length-index-1)

### Searching:

G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
								T	A	T	G	T	G			

  
 G C A A T G C C T A T G T G A C C  
                     T A T G T G           +4

Output: Pattern found at index 8.

## KMP Algorithm (DP):

### Preprocessing:

- Construct an array p of size m (same as the size of pattern) which is used to skip characters while matching.
- The array holds indices of the longest proper prefix which is also suffix. A proper prefix is a prefix with whole string not allowed.
- $p[i]$  = the longest proper prefix of  $b[0..i]$  which is also a suffix of  $b[0..i]$ .

### Searching:

- Initially, the window of text is taken from the starting index of text (size same as that of the pattern).
- We start the comparison of pattern with characters of the current window of text both from the left side.
- We keep matching characters of the text window and pattern and keep incrementing the indices till we find a mismatch.
- When we see a mismatch, we use p array to find the next index from where the searching should progress again.

- If we reach the end of the pattern without a mismatch, then print the occurrence and begin the next iteration.

### Example:

DNA: G C A A T G C C T A T G T G A C C  
 Pattern: T A T G T G

### Preprocessing:

Table p (size of pattern):

<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>
----------	----------	----------	----------	----------	----------

Index:            0            1            2            3            4            5

### Searching:

↓ **i**  
 G C A A T G C C T A T G T G A C C  
 ↓ **j**  
 T A T G T G

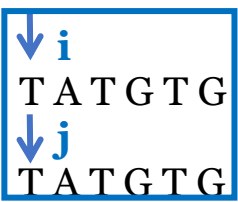
Comparing characters at i and j after 4 steps:

↓ **i**  
 G C A A T G C C T A T G T G A C C  
 ↓ **j**  
 T A T G T G  
 ↓ **i**  
 G C A A T G C C T A T G T G A C C  
 ↓ **j**  
 T A T G T G

j is set to p[j-1] i.e. 0. And since j becomes 0 and a[j] != b[j] we increase i.

↓ **i**  
 G C A A T G C C T A T G T G A C C  
 ↓ **j**  
 T A T G T G

after two steps:



G C A A T G C C T A T G T G A C C  
 T A T G T G

## Implementation and Results:

### 1.Brute Force:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    string a,b;
    int i,j;
    cin>>a>>b;
    for(i=0;i<=a.length()-b.length();i++){
        int flag=1;
        for(j=0;j<b.length();j++){
            if(b[j]!=a[i+j]){
                flag=-1;
                break;
            }
        }
        if(flag==1)
            cout<<endl<<"Pattern    found    at    position
"<<i+1<<endl;
    }
    return 0;
}
```



## 2.Boyer Moore Algorithm :

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    string a,b;int i,j;
    cin>>a>>b;
    vector <int> p(26,b.length());
    for(i=0;i<b.length()-1;i++)
        p[b[i]-'A']=b.length()-i-1;
    p[b.length()-1]=b.length();
    j=b.length()-1;
    for(i=b.length()-1;i<a.length()&&i>=0&&j>=0;){
        int k=i;
        while(a[k]==b[j]&&j>=0&&k>=0){
            k--;
            j--;
        }
        if(j== -1) {
            cout<<"Pattern found at "<<k+1<<endl;
            i++;
        }
        else i=k+p[a[k]-'A'];
        j=b.length()-1;
    }
    return 0;
}
```

### 3.KMP Algorithm:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    string a,b;
    int i,j;
    cin>>a>>b;
    vector <int> p(b.size());
    j=0;
    p[0]=0;
    for(i=1;i<b.length();){
        if(b[i]==b[j]){
            p[i]=j+1;
            j++;
            i++;
        }
        else{
            while(b[j]!=b[i]&& j!=0)
                j=p[j-1];
            if(b[j]!=b[i]){
                p[i]=0;
                i++;
            }
        }
    }
}
```

```

//search
    j=0;
    for(i=0;i<a.length();)
    {
        if(a[i]==b[j]) {i++;j++;}
        else
        {
            while(b[j]!=a[i]&&j!=0)
                j=p[j-1];
            if(a[i]!=b[j])
                i++;
        }
        if(j==b.length())
        {
            cout<<"Pattern found at "<<i-b.length()+1<<endl;
            j=0;
        }
    }
    return 0;
}

```

# Results:

## 1.Brute Force:

```
shravya@shravya-Vostro-3478: ~/Desktop
File Edit View Search Terminal Help
shravya@shravya-Vostro-3478:~/Desktop$ g++ brute.cpp
shravya@shravya-Vostro-3478:~/Desktop$ ./a.out
AABAACAADAABAABA
AABA
Pattern found at position 1
Pattern found at position 10
Pattern found at position 13
shravya@shravya-Vostro-3478:~/Desktop$
```

## 2.Boyer Moore algorithm:

```
shravya@shravya-Vostro-3478: ~/Desktop
File Edit View Search Terminal Help
shravya@shravya-Vostro-3478:~/Desktop$ g++ bm.cpp
shravya@shravya-Vostro-3478:~/Desktop$ ./a.out
AABAACAADAABAABA
AABA
pattern found at 1
pattern found at 10
pattern found at 13
shravya@shravya-Vostro-3478:~/Desktop$
```

## 3.KMP algorithm:

```
shravya@shravya-Vostro-3478: ~/Desktop
File Edit View Search Terminal Help
shravya@shravya-Vostro-3478:~/Desktop$ g++ kmp.cpp
shravya@shravya-Vostro-3478:~/Desktop$ ./a.out
AABAACAADAABAABA
AABA
Pattern found at 1
Pattern found at 10
Pattern found at 13
shravya@shravya-Vostro-3478:~/Desktop$
```

## **Complexity Analysis:**

### **1.Brute Force:**

Step	Time
For i: 0 to n-m;	$O(n)$
For j: 0 to m	$O(m)$
Total	$O(n*m)$
Overall Space Complexity	$O(1)$

### **2.Boyer Moore Algorithm:**

Step	Best	Worst
Pre Processing (Finding values of vector p)	$O(m)$	$O(m)$
Searching	$O(n+m)$	$O(m*n)$
Total	$O(n+m)$	$O(m*n)$
Overall Space Complexity	$O(1)$	

### **3.KMP Algorithm:**

Step	Time
Pre Processing (Finding values of vector p)	$O(m)$
Searching	$O(n+m)$
Total	$O(n+m)$
Overall Space Complexity	$O(m)$

## Comparison of Algorithms:

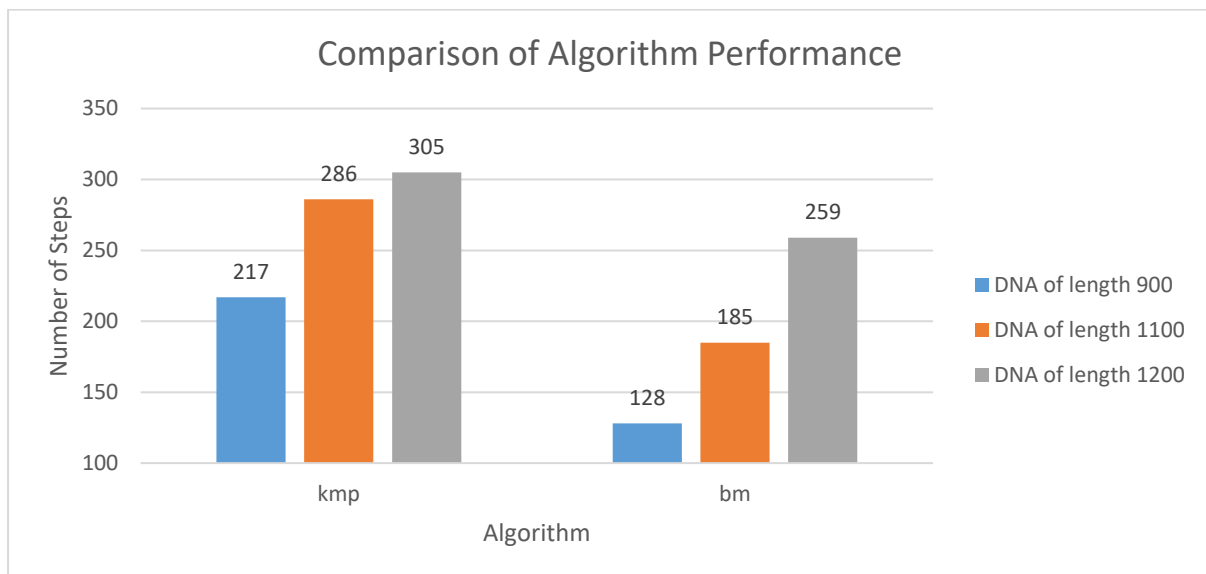
```
shravya@shravya-Vostro-3478: ~/Desktop
File Edit View Search Terminal Help
shravya@shravya-Vostro-3478:~/Desktop$ g++ brute.cpp
shravya@shravya-Vostro-3478:~/Desktop$ ./a.out<in.txt

Pattern found at position 352
Pattern found at position 810
Pattern found at position 5314
Pattern found at position 5635
Pattern found at position 8535
Pattern found at position 8994
Pattern found at position 9613
13291Steps
shravya@shravya-Vostro-3478:~/Desktop$ g++ kmp.cpp
shravya@shravya-Vostro-3478:~/Desktop$ ./a.out<in.txt
Pattern found at 352
Pattern found at 810
Pattern found at 5314
Pattern found at 5635
Pattern found at 8535
Pattern found at 8994
Pattern found at 9613
2461 Steps
shravya@shravya-Vostro-3478:~/Desktop$ g++ bm.cpp
shravya@shravya-Vostro-3478:~/Desktop$ ./a.out<in.txt
Pattern found at 352
Pattern found at 810
Pattern found at 5314
Pattern found at 5635
Pattern found at 8535
Pattern found at 8994
Pattern found at 9613
1428 Steps
shravya@shravya-Vostro-3478:~/Desktop$
```

For strings of considerably large length performance can be compared as:

Brute Force < KMP Algorithm < Boyer Moore Algorithm

Further analysis was made to compare the performance of the algorithms. Three test cases each with DNA of length 900,1100 and 1200 were used.



## **Applications of Pattern Matching:**

- Locating DNA sequences.
- Fingerprint assessment.
- Soil patterns reporting.
- Retinal blood vessel assessment.
- Diseases caused by genetic factors are also detected.
- The intensity and existence of a disease can be predicted with the number of occurrences of a particular sequence in the DNA.

## **References:**

- <https://pdfs.semanticscholar.org/2a08/3422a388c2773668a8808b33207b8cf37241.pdf>
  - <http://ijsetr.com/uploads/625413IJSETR2868-162.pdf>
  - [https://en.wikipedia.org/wiki/String-searching\\_algorithm](https://en.wikipedia.org/wiki/String-searching_algorithm)
- Website used for generating test cases: <http://www.unit-conversion.info/texttools/random-string-generator/>