

**PROJECT – 3**

**Pattern Matching Algorithms**

**DEPARTMENT OF COMPUTER SCIENCE**

**ITCS-6114**

**Algorithms and Data Structures**

**Submitted to** **Dewan T. Ahmed, Ph.D.**

**Submitted By:**

**Yash kanodia Dipesh Chakrani**

**801136232 801080983**

# Introduction

# The Pattern Searching algorithms are now and again also referred to as String Searching Algorithms and are regarded as a section of the String algorithms. These algorithms are beneficial in the case of searching a string within any other string. As with most algorithms, the main considerations for string searching are speed and efficiency.

Following is the listing of sample matching strategies that we have implemented:

1. Brute Force
2. Boyer-Moore-Horspool Algorithm
3. Knuth-Morris-Pratt Algorithm

# Brute Force

The principle of brute-force search is quite simple. The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found. We keep comparing the characters from left to right (it is crucial because faster approaches work differently). The algorithm checks whether the actual character in the text matches the given character in the pattern.  Whenever a mismatch is found, the remaining character comparisons for that substring are dropped and the next substring can be selected immediately.

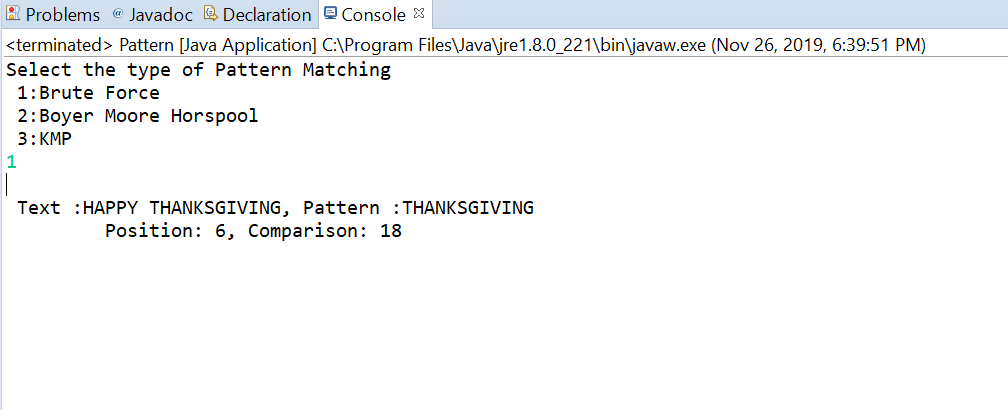
**Time complexity:**

In worst case, Brute Force Algorithm runs in O(nm) time where n is the length of text and m is the length of pattern.

**Data Structure Used:**

An array is used to store the return values in Brute Force Algorithm.

**Sample output:**



# Boyer-Moore-Horspool Algorithm

Boyer-Moore-Horspool is an algorithm for finding substrings into strings. This algorithm compares each characters of substring to find a word or the same characters into the string. When characters do not match, the search jumps to the next matching position in the pattern by the value indicated in the shift Table. Like Boyer–Moore, Boyer–Moore–Horspool preprocesses the pattern to produce a table containing, for each symbol in the [alphabet](https://en.wikipedia.org/wiki/Alphabet_(formal_languages)), the number of characters that can safely be skipped. The Bad character shift used in the Boyer-Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful.

**Time Complexity:**

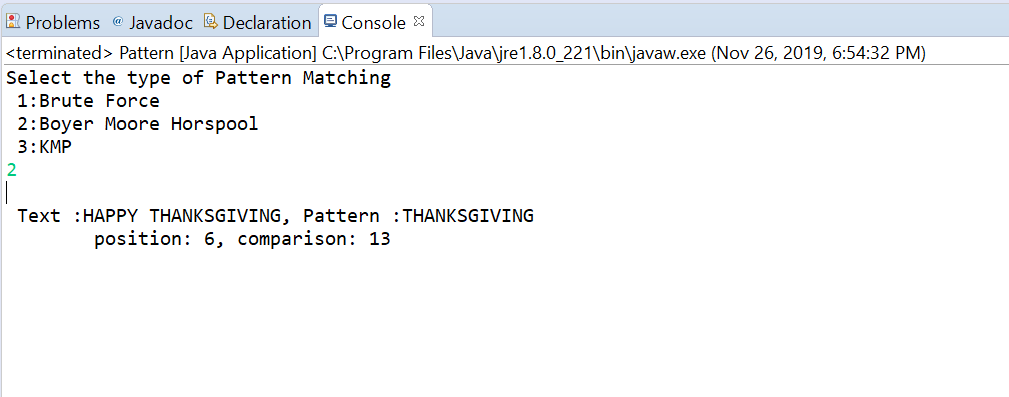
Preprocessing time of Horspool Algorithm is O(S+m). The worst case runtime of this algorithm is O(nm).

The search time and best case runtime of this algorithm is O(n/m) where n is the length of the text and m is the length of the pattern.

**Data Structure Used:**

Horspool Algorithm is using an array data structure to store the position and numbers of comparisons used.

**Sample Output:**



# III. Knuth-Morris-Pratt Algorithm

# 

Knuth Morris Pratt (KMP) is an algorithm, which checks the characters from left to right. When a pattern has a sub-pattern appears more than one in the sub-pattern, it uses that property to improve the time complexity, also for in the worst case. It never compares the characters that are already been compared. It uses one failure table that can be built by comparing the length of prefix and suffix of the pattern.

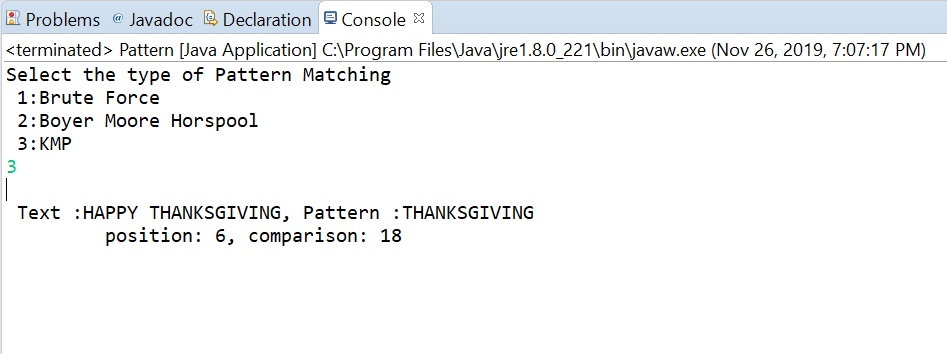
**Data Structure Used:**

An array data structure is used for storing the failure table information also the position and comparisons of the pattern.

**Time complexity:**

The Worst case runtime of KMP algorithm is O(n).

**Sample Output:**



1. Code:

**import** java.util.Scanner;

**public** **class** Pattern {

**public** **static** **int**[] BruteForce(String txt, String pat)

{

**int** arr[] = **new** **int**[2];

**int** a = txt.length();

**int** b = pat.length();

**int** c = 0;

**for** (**int** i = 0; i <= a - b; i++)

{

**int** j = 0;

**while** (j < b)

{

c++;

**if** (txt.charAt(i + j) == pat.charAt(j))

{

j++;

}

**else**

{

**break**;

}

**if** (j == b) {

arr[0] = i;

arr[1] = c;

**return** arr;

}

}

}

**return** **new** **int**[] {-1,c};

}

**public** **static** **int**[] BadSymbol(String pat) {

**int** b = pat.length();

**int** table[] = **new** **int**[255];

**for** (**int** i = 0; i < 255; i++) {

table[i] = b;

}

**for** (**int** j = 0; j < b - 1; j++) {

table[pat.charAt(j)] = b - 1 - j;

}

**return** table;

}

**public** **static** **int**[] BMHorspoolMatching(String txt, String pat) {

**int**[] position\_and\_comparisions = **new** **int**[2];

**int** c = 0;

**int**[] table = *BadSymbol*(pat);

**int** m = pat.length();

**int** n = txt.length();

**int** i = m - 1;

**while** (i < n) {

**int** k = 0;

**while** (k < m) {

c++;

**if** (pat.charAt(m - 1 - k) != txt.charAt(i - k))

{

**break**;

}

**else**

{

k++;

}

}

**if** (k != m)

{

i = i + table[txt.charAt(i)];

}

**else**

{

position\_and\_comparisions[0] = i - m + 1;

position\_and\_comparisions[1] = c;

**return** position\_and\_comparisions;

}

}

**return** **new** **int**[] { -1, c };

}

**public** **static** **int**[] FailureFunction(String pat) {

**int** m = pat.length();

**int** i = 1;

**int** j = 0;

**int**[] failure = **new** **int**[m];

failure[0] = 0;

**while** (i < m) {

**if** (pat.charAt(j) == pat.charAt(i))

{

failure[i] = j + 1;

i++;

j++;

}

**else** **if** (j > 0)

{

j = failure[j - 1];

}

**else**

{

failure[i] = 0;

i++;

}

}

**return** failure;

}

**public** **static** **int**[] KMPMatch(String txt, String pat) { //KMP matching method

**int**[] failure = *FailureFunction*(pat);

**int**[] position\_and\_comparisions = **new** **int**[2];

**int** i = 0;

**int** j = 0;

**int** n = txt.length();

**int** m = pat.length();

**int** c = 0;

**while** (i < n) {

c++;

**if** (pat.charAt(j) == txt.charAt(i))

{

**if** (j == m - 1)

{

position\_and\_comparisions[0] = i - m + 1;

position\_and\_comparisions[1] = c;

**return** position\_and\_comparisions;

}

i++;

j++;

}

**else** **if** (j > 0)

{

j = failure[j - 1];

}

**else**

{

i++;

}

}

**return** **new** **int**[] { -1, c };

}

**public** **static** **void** main(String[] args)

{

String text = "ABACAABACCABACABAABB";

String pattern = "ABACAB";

Scanner scan=**new** Scanner(System.***in***);

System.***out***.println("Select the type of Pattern Matching \n 1:Brute Force \n 2:Boyer Moore Horspool \n 3:KMP ");

**int** userInput= scan.nextInt();

**switch**(userInput)

{

**case** 1:

**int**[] BruteForce = *BruteForce*(text,pattern);

System.***out***.println("\n Text :"+text+", Pattern :"+pattern);

System.***out***.print("\t Position: "+BruteForce[0]+", "+"Comparison: "+BruteForce[1]);

**break**;

**case** 2:

**int**[] BMHorspool = *BMHorspoolMatching*(text, pattern);

System.***out***.println("\n Text :"+text+", Pattern :"+pattern);

System.***out***.print("\tposition: "+BMHorspool[0]+", "+"comparison: "+BMHorspool[1]);

**break**;

**case** 3:

**int**[] KMP = *KMPMatch*(text, pattern);

System.***out***.println("\n Text :"+text+", Pattern :"+pattern);

System.***out***.print("\t position: "+KMP[0]+", "+"comparison: "+KMP[1]);

**break**;

}

}

}

1. Comparison:

Case I:

Text: ABC ABCDAB ABCDABCDABDE

Pattern: ABCDABD

Case II:

Text: FINDINAHAYSTACKNEEDLEIN

Pattern: NEEDLE

Case III:

Text: jim\_saw\_me\_in\_a\_barbershop

Pattern: barber

Case IV:

Text: bard loved bananas

Pattern: baobab

Case V:

Text: bess\_knew\_about\_baobabs

Pattern: baobab

Case VI:

Text: ABABABAEABABACBD

Pattern: ABABACB

Case VII:

Text: abacaabaccabacabaabb

Pattern: abacab

Case VIII:

Text: TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT

Pattern: TCCTATTCTT

Case IX:

Text: ABABABAABACABACABC

Pattern: ABACABC

Case X:

Text: CBBAABAABBCABAAABBBABBAAB

Pattern: ABBAAB

|  |  |  |  |
| --- | --- | --- | --- |
|  | Brute-Force | BMH | KMP |
| I | 39 | 10 | 26 |
| II | 23 | 11 | 23 |
| III | 22 | 12 | 22 |
| IV | 17 | 4 | 20 |
| V | 24 | 13 | 24 |
| VI | 29 | 11 | 19 |
| VII | 28 | 15 | 19 |
| VIII | 56 | 38 | 52 |
| IX | 35 | 13 | 23 |
| X | 39 | 31 | 33 |