UNC CHARLOTTE
College of Computing and Informatics

Algorithms & Data Structures Project 3

Pattern Matching Algorithms

Submitted To:

Dr Dewan Tanvir Ahmed

Submitted By:

Ankit Pandita

Avijit Jaiswal

Jinraj Jain

1.1 Brute force algorithm

It is one of the early pattern matching algorithm that consists of systematically comparing all possible values for the solution and checking whether each value satisfies the problem's statement. It starts matching pattern from left to right.

Brute force is simple to implement, and will always find a solution if it exist. brute-force search is typically used when the problem size is limited.

The time complexity of brute force is O(n*m) . So, if we were to search for a pattern of ‘n’ characters in a text of ‘m’ characters using brute force, it would take us n * m tries.

Algorithm:

Applying brute-force search to a given pattern , These procedures should take as a pattern *P* for the particular instance of the problem from text T , we should do the following to solve the given problem:

```
Algorithm BruteForceMatch(T, P)
     Input text T of size n and pattern P of size m
     Output starting index of a substring of T equal to
               P or -1 if no such substring exists
for  i ← 0 to n - m
     { test shift i of the pattern }
     j ← 0
     while j < m and T[i + j] = P[j]
               j ← j + 1
               if  j = m
     return  i {match at i}
else
          break while loop {mismatch}
return  -1 {no match anywhere}
```

1.2 Knuth Morris-Pratt (KMP)

It is a pattern matching algorithm which starts comparing from right to left. It is a better approach than brute force algorithm. 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.

The time complexity of KMP is O(n).

The main idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window.

We take advantage of this information to avoid matching the characters that we know will anyway match. Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.

- The failure function *F*(*j*) is defined as the size of the largest prefix of P[0..j] that is also a suffix of P[1..j]

Algorithm:

```
Algorithm KMPMatch(T, P)
     F ← failureFunction(P)
     i ← 0
     j ← 0
     while i < n
     if T[i] = P[j]
          if  j = m - 1
               return  i - j { match }
          else
               i ← i + 1
               j ← j + 1
     else
          if  j > 0
               j ← F[j - 1]
          else
               i ← i + 1
return  -1 { no match }
```

1.3 Boyer Moore Horspool Algorithm

The Boyer–Moore–Horspool algorithm or Horspool's algorithm is an algorithm for finding substrings in strings. It was published by Nigel Horspool in 1980 as SBM.

It is a simplification of the Boyer–Moore string search algorithm which is related to the Knuth–Morris–Pratt algorithm.

Like Boyer–Moore, Boyer–Moore–Horspool preprocesses the pattern to produce a table containing, for each symbol in the alphabet, the number of characters that can safely be skipped.

The algorithm performs best with long needle strings, when it consistently 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.

> preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
>
> always makes a shift based on the text’s character c aligned with the last character in the pattern according to the shift table’s entry for c

Shift sizes can be precomputed by the formula by scanning pattern before search begins and stored in a table called shift table.

In the worst-case the time complexity of the Boyer-Moore-Horspool algorithm is O(mn), where m is the length of the substring and n is the length of the string.

The average time is O(n). In the best case, the performance is sub-linear, and is, in fact, identical to Boyer-Moore original implementation.

Algorithm:

```
function preprocess(pattern)
    T ← new table of 256 integers
    for i from 0 to 256 exclusive
        T[i] ← length(pattern)
    for i from 0 to length(pattern) - 1 exclusive
        T[pattern[i]] ← length(pattern) - 1 - i
    return T

function same(str1, str2, len)
```

```
    i ← len - 1
    while str1[i] = needle[i]
        if i = 0
            return true
        i ← i - 1
    return false


function search(needle, haystack)
    T ← preprocess(needle)
    skip ← 0
    while length(haystack) - skip ≥ length(needle)
        haystack[skip:] -- substring starting with "skip".
&haystack[skip] in C.
        if same(haystack[skip:], needle, length(needle))
            return skip
            i ← i - 1
        skip ← skip + T[haystack[skip + length(needle) - 1]]
    return not-found
```

2.1 Program Structure

The programming language used for developing the code is Java.

The data structure used is Arrays for Brute Force and Knuth-Morris-Pratt algorithms and Hashmap and Arrays for Horspool algorithm.

The program consists of four main classes:

Main.java : This file contains the main driving code for the program.

BruteForce.java : This file contains the implementation for Brute Force algorithm.

BMHorspool.java : This file contains the implementation for Boyer-Moore-Horspool algorithm.

KnuthMorrisPratt.java : This file contains the implementation for Knuth-Morris-Pratt algorithm.

<u>RESULT TABLE:</u>

S no	Input	Brute force [comparisions]	Horspool [comparisions]	KMP Algorithm [comparisions]
1	Text: Hi! My name is Ankit Pandita. Pattern: Ankit	20	8	20
2	Text: Jinraj is cooking. Pattern: cook	14	8	14
3	Text: Avajit is talking Pattern: BOABAB	12	2	17
4	Text: Jim saw me in a barbershop Pattern: barber	22	12	22
5	Text: BARD LOVES BANANAS Pattern: BANANA	19	8	18
6	Text: bess new about baobabs Pattern: baobab	23	9	23
7	Text: ABABEABABAABABABAEABABACBDHDSDKWABABATWI Pattern: ABABACB	54	17	34
8	Text: AAAABBABABBBBABBBBBBBABBBBABABBBBAAABABABABABABAA Pattern: BBBBABBBB	22	18	21
9	Text: I went for a run Pattern: run	17	8	17
10	Text: Merry Christmas and A Very Happy New Year Pattern: Christmas	15	10	15

2.2 SOURCE CODE

Main.java

```
package edu.uncc.cci.algods;

public class Main {

    public static void main(String[] args) {
        String[] text = new String[10];
        String[] pattern = new String[10];

        text[0] = "Hi! My name is Ankit Pandita.";
        pattern[0] = "Ankit";

        text[1] = "Jinraj is cooking.";
        pattern[1] = "cook";

        text[2] = "Avajit is talking";
        pattern[2] = "BOABAB";

        text[3] = "Jim saw me in a barbershop";
        pattern[3] = "barber";

        text[4] = "BARD LOVES BANANAS";
        pattern[4] = "BANANA";

        text[5] = "bess new about baobabs";
        pattern[5] = "baobab";

        text[6] = "ABABEABABAABABABAEABABACBDHDSDKWABABATWI";
        pattern[6] = "ABABACB";


        text[7] =
"AAAABBABABBBBABBBBBBBABBBBABABBBBAAABABABABABABABAA";
        pattern[7] = "BBBBABBBB";


        text[8] = "I went for a run";
        pattern[8] = "run";


        text[9] = "Merry Christmas and A Very Happy New Year";
        pattern[9] = "Christmas";



        for (int i = 0; i < text.length; i++) {
            System.out.println("Text: " + text[i]);
            System.out.println("Pattern: " + pattern[i]);
            System.out.println();


            BruteForce bruteForce = new BruteForce();
            int[] resultArray = bruteForce.naiveAlgo(text[i],
pattern[i]);
            int bruteForceResult = resultArray[0];
            int comparisonCountBruteForce = resultArray[1];
            System.out.println("Brute-Force algorithm");
            System.out.println(bruteForceResult == -1 ? "No
match found" : "Match start index: " + bruteForceResult);
            System.out.println("Number of comparisons: " +
comparisonCountBruteForce);
            System.out.println();


            BMHorspool bmHorspool = new BMHorspool();
            resultArray = bmHorspool.horspoolAlgo(text[i], 
pattern[i]);
            int horspoolResult = resultArray[0];
            int comparisonCountHorsepool = resultArray[1];
            System.out.println("Boyer-Moore-Horspool 
algorithm");
            System.out.println(horspoolResult == -1 ? "No 
match found" : "Match start index: " + horspoolResult);
            System.out.println("Number of comparisons: " + 
comparisonCountHorsepool);
            System.out.println();


            KnuthMorrisPratt knuthMorrisPratt = new 
KnuthMorrisPratt();
            resultArray = knuthMorrisPratt.kmpAlgo(text[i], 
pattern[i]);
            int kmpResult = resultArray[0];
            int comparisonCountKMP = resultArray[1];
            System.out.println("Knuth-Morris-Pratt 
algorithm");
            System.out.println(kmpResult == -1 ? "No match 
found" : "Match start index: " + kmpResult);
            System.out.println("Number of comparisons: " + 
comparisonCountKMP);
            System.out.println("------------------------------
-----------------------");
            System.out.println();
        }
    }
}
```

BruteForce.java

```
package edu.uncc.cci.algods;

public class BruteForce {
    public int[] naiveAlgo(String text, String pattern) {
        int comparisionCount = 0;
        for (int i = 0; i <= text.length() - pattern.length();
i++) {
            int count = 0;
            for (int j = 0; j < pattern.length(); j++) {
                comparisionCount++;
                if (text.charAt(i + j) == pattern.charAt(j))
                    count++;
                else break;
            }
            if (count == pattern.length())
                return new int[]{i, comparisionCount};
        }
        return new int[]{-1, comparisionCount};
    }
}
```

BMHorspool.java

```
package edu.uncc.cci.algods;

import java.util.HashMap;

public class BMHorspool {
    public int[] horspoolAlgo(String text, String pattern) {
        HashMap<Character, Integer> shiftTable = new
HashMap<>();
        int patternLength = pattern.length();
        for (int i = 0; i < patternLength; i++) {
            if (!shiftTable.containsKey(pattern.charAt(i)))
                shiftTable.put(pattern.charAt(i),
patternLength);
        }
        for (int i = 0; i <= patternLength - 2; i++) {
            int charIndex =
pattern.lastIndexOf(pattern.charAt(i));
            if (charIndex < patternLength - 1)
                shiftTable.replace(pattern.charAt(i),
patternLength - 1 - charIndex);
            else
                shiftTable.replace(pattern.charAt(i),
patternLength - 1 - i);
        }
        int i = patternLength - 1;
        int comparisionCount = 0;
        while (i < text.length()) {
            int k = 0;
            while (k < patternLength) {
                if (text.charAt(i - k) ==
pattern.charAt(patternLength - 1 - k)) {
                    comparisionCount++;
                    k++;
                    if (k == patternLength)
                        return new int[]{i - patternLength +
1, comparisionCount};
                } else {
                    comparisionCount++;
                    if
(shiftTable.containsKey(text.charAt(i)))
                        i = i +
shiftTable.get(text.charAt(i));
                    else i = i + patternLength;
                    break;
                }
            }
        }
        return new int[]{-1, comparisionCount};
    }
}
```

KnuthMorrisPratt.java

```
package edu.uncc.cci.algods;

public class KnuthMorrisPratt {
    public int[] kmpAlgo(String text, String pattern) {
        int[] failureTable = generateFailureTable(pattern);
        int j = 0;
        int comparisionCount = 0;
        for (int i = 0; i < text.length(); i++) {
            while (j > 0 && text.charAt(i) !=
pattern.charAt(j)) {
                comparisionCount++;
                j = failureTable[j - 1];
            }
            if (text.charAt(i) == pattern.charAt(j)) {
                comparisionCount++;
                j++;
                if (j == pattern.length())
                    return new int[]{i - (j - 1), 
comparisionCount};
            } else
                comparisionCount++;
        }
        return new int[]{-1, comparisionCount};
    }


    public int[] generateFailureTable(String pattern) {
        int patternLength = pattern.length();
        int[] failureTable = new int[patternLength];
        failureTable[0] = 0;
        for (int i = 1; i < patternLength; i++) {
            int k = failureTable[i - 1];
            while (k > 0 && pattern.charAt(i) != 
pattern.charAt(k))
                k = failureTable[k - 1];
            if (pattern.charAt(i) == pattern.charAt(k))
                k++;
            failureTable[i] = k;
        }
        return failureTable;
    }
}
```