**ADVANCED ALGORITHM ASSIGNMENT**

**SUBJECT CODE: 17CS554**

**TOPIC:**

**String-Matching Algorithms**

**SUBMITTED BY:**

**VARUN S**

**1MJ17CS172**



**DEPARTMENT OF COMPUTER SCIENCE**

**2019-20**

String-Matching Algorithm

**Introduction**

String-searching algorithms also known as string-matching algorithms are an important class of string algorithms that try to find where one or several strings knowns as patterns are found within a larger string or text.

**INPUT:** A A B A A C A A D A A B A A B A

**PATTERN:** A A B A

A A B A A A B A

A A B A A C A A D A A B A A B A

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

A A B A

Thus, we can say that the pattern is found at

index 0, 9, 12.

**AIM**

Given a text and pattern, for which the pattern is a substring of the given text, find the pattern in the given string and print the index at which the pattern can be found.

**SCOPE**

The given algorithms can be used in multiple fields. Some of them are as follows:

1) Spell check

2) Spam filter

3) Search engines

4) Plagiarism check

5) Bioinformatics

6) Digital forensics

**LANGUAGE USED**

Java is a general purpose programming language that is class base, object oriented and designed to have as few implementation dependencies as possible. It is intended to let application developers write once, run anywhere, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

**ALGORITHMS USED**

1) NAIVE ALGORITHM

Given a text input[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

INPUT:

input[] = “AABAACAADAABA”

pat[] = “AABA”

OUTPUT:

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

BEST CASE:

The best case occurs when the first character of the pattern is not present in text at all.

The time complexity: O(n)

WORST CASE:

There are 2 cases:

a) When all characters of the text and pattern are the same.

b) When only the last character is different.

The time complexity: O(m\*(n-m+1))

2) KNUTH MORRIS PRATT(KMP) ALGORITHM

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

INPUT:

input[] = “AABAACAADAABA”

pat[] = “AABA”

OUTPUT:

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

MATCHING OVERVIEW:

input = “AAAAABAAABA”

pat = “AAAA”

We compare input with patterns

input = A A A A A B A A A B A

pat = A A A A

We find a match.

We now compare a next window of input with pat.

input = A A A A A B A A A B A

pat = A A A A

This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. We skipped the first 3 characters since we knew they’d match anyway.

3) Rabin Karp Algorithm

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

INPUT:

input[] = “AABAACAADAABA”

pat[] = “AABA”

OUTPUT:

Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

In Rabin Karp Algorithm, the pattern slides one by one. In RK, the hash value of the pattern is matched with the hash value of the current substring. If the hash values match only then it starts matching individual characters.

The hash value for the following strings are calculated for the pattern itself and for all the substrings of text of length m.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say hash(txt[s+1..s+m]) must be efficiently computable from hash(txt[s..s+m1]) and txt[s+m] i.e., hash(txt[s+1.,s+m])= rehash(txt[s+m], hash(txt[s..s+m-1])) and rehash must be O(1) operation.

d: Number of characters in the alphabet

q: A prime number h: d^(m-1)

h: d^(m-1)

IMPLEMENTATION

AlgProj.java

**import** java.util.\*;

**public** **class** AlgProj {

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

{

String input, pat;

**int** ch;

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

**while**(**true**)

{

System.***out***.println("\n\t \t \t PATTERN MATCHING \t \t \t");

System.***out***.println("\nEnter the string\n");

input = sc.next();

System.***out***.println("\nEnter the pattern\n");

pat = sc.next();

System.***out***.println("\n1. Naive");

System.***out***.println("\n2. KMP");

System.***out***.println("\n3. Rabin Karp");

Naive n = **new** Naive();

kmp k = **new** kmp();

rabinkarp rk = **new** rabinkarp();

System.***out***.println("\n Enter your choice");

ch = sc.nextInt();

**switch**(ch)

{

**case** 1: n.search(input,pat);

**break**;

**case** 2: k.KMPSearch(pat, input);

**break**;

**case** 3: rk.search(pat, input);

**break**;

}

}

}

}

Naive.java

**public** **class** Naive {

**public** **void** search(String input, String pat)

{

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

**int** N = input.length();

**int** i = 0;

**while** (i <= N - M)

{

**int** j;

**for** (j = 0; j < M; j++)

**if** (input.charAt(i + j) != pat.charAt(j))

**break**;

**if** (j == M)

{

System.***out***.println("Pattern found at index "+i);

i = i + M;

}

**else** **if** (j == 0)

i = i + 1;

**else**

i = i + j;

}

}

}

kmp.java

**public** **class** kmp {

**void** KMPSearch(String pat, String input)

{

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

**int** N = input.length();

**int** lps[] = **new** **int**[M];

**int** j = 0; // index for pat

computeLPSArray(pat, M, lps);

**int** i = 0; // index for input

**while** (i < N) {

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

j++;

i++;

}

**if** (j == M) {

System.***out***.println("Found pattern "

+ "at index " + (i - j));

j = lps[j - 1];

}

**else** **if** (i < N && pat.charAt(j) != input.charAt(i)) {

**if** (j != 0)

j = lps[j - 1];

**else**

i = i + 1;

}

}

}

**void** computeLPSArray(String pat, **int** M, **int** lps[])

{

**int** len = 0;

**int** i = 1;

lps[0] = 0;

**while** (i < M) {

**if** (pat.charAt(i) == pat.charAt(len)) {

len++;

lps[i] = len;

i++;

}

**else**

{

**if** (len != 0) {

len = lps[len - 1];

}

**else**

{

lps[i] = len;

i++;

}

}

}

}

}

rabinkarp.java

**public** **class** rabinkarp {

**public** **final** **static** **int** ***d*** = 256;

**public** **void** search(String pat, String input)

{

**int** q = 101;

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

**int** N = input.length();

**int** i, j;

**int** p = 0; // hash value for pattern

**int** t = 0; // hash value for input

**int** h = 1;

**for** (i = 0; i < M-1; i++)

h = (h\****d***)%q;

**for** (i = 0; i < M; i++)

{

p = (***d***\*p + pat.charAt(i))%q;

t = (***d***\*t + input.charAt(i))%q;

}

**for** (i = 0; i <= N - M; i++)

{

**if** ( p == t )

{

**for** (j = 0; j < M; j++)

{

**if** (input.charAt(i+j) != pat.charAt(j))

**break**;

}

**if** (j == M)

System.***out***.println("Pattern found at index " + i);

}

**if** ( i < N-M )

{

t = (***d***\*(t - input.charAt(i)\*h) + input.charAt(i+M))%q;

**if** (t < 0)

t = (t + q);

}

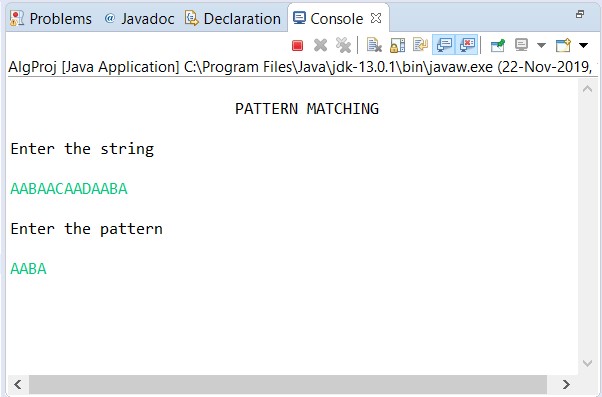
}

}

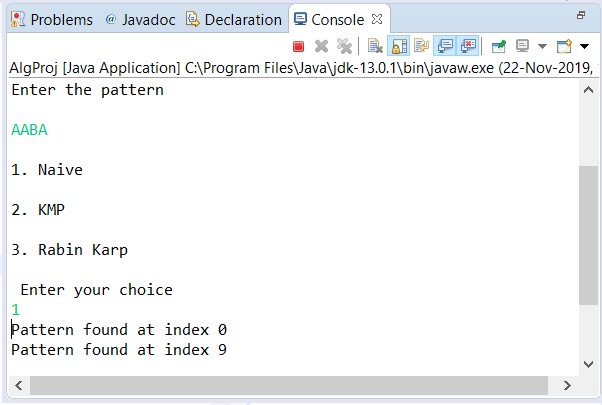
}

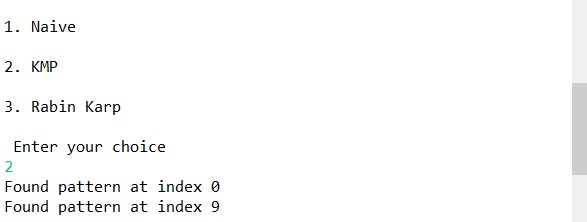
OUTPUT

1) String and Pattern



2) Naive Algorithm



3) KMP Algorithm

4) Rabin Karp Algorithm

