**STRING MATCHING ALGORITHMS**

**INTRODUCTION**

The Pattern Searching algorithms are sometimes also referred to as

String Searching Algorithms and are considered as a part of the String algorithms. These algorithms are useful in the case of searching a string within another string.



**PROBLEM STATEMENT**

Given a text and a pattern which is a substring of the given text ,objective is to

return the index of occurence of pattern in the text.

**SCOPE OF THE PROJECT**

Program developed can be used in various fields as follows :

**STRING MATCHING ALGORITHMS**

**1.Spell checkers**

**2.Spam filters/spam detection systems**

**3.Intrusion detection system**

**4.Search engines/Content searching in large databases**

**5.Plagiarism detection**

**6.Bioinformatics/DNA Sequencing**

**7.Digital forensics**

**LANGUAGE USED:**

Java**is an object-oriented programming language**developed**by James Gosling and colleagues at Sun Microsystems in the early 1990s.**Java**was designed to have the look and feel of the C++ programming language, but is simpler to use and enforces an object-oriented programming model.**Java**can be**used**to create complete applications that may run on a single computer or be distributed among servers and clients in a network.**

**ALGORITHMS USED**

**NAIVE STRING MATCHING ALGORITHM:**

Given a text *txt[0..n-1]*and a pattern

*pat[0..m-1],function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.*

**Examples:**

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"

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.

txt[] = "AABCCAADDEE"; pat[] = "FAA";

The number of comparisons in best case is O(n).

**Worst case:**

The worst case of Naive Pattern Searching occurs in following scenarios. 1) When all characters of the text and pattern are same.

txt[] = "AAAAAAAAAAAAAAAAAA"; pat[] = "AAAAA";

2) Worst case also occurs when only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAA"; pat[] = "AAAAA";

**KMP (KNUTH MORRIS PRATT) ALGORITHM:**

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic 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.

**Matching Overview**

txt = "AAAAABAAABA" pat = "AAAA"

We compare first window of **txt**with **pat**txt = "**AAAA**ABAAABA"

pat = "**AAAA**" [Initial position]

We find a match. This is same as [Naive String Matching.](https://www.geeksforgeeks.org/searching-for-patterns-set-1-naive-pattern-searching/)

In the next step, we compare next window of **txt**with **pat**. txt = "**AAAAA**BAAABA"

pat = "**AAAA**" [Pattern shifted one position]

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. Since we know first three characters will anyway match, we skipped matching first three characters.

**Need of Preprocessing?**

An important question arises from the above explanation,

how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array

lps[] that tells us the count of characters to be skipped.

**Preprocessing Overview:**

∙KMP algorithm preprocesses pat[] and constructs an auxiliary **lps[]**of size m (same as size of pattern) which is used to skip characters while matching.

∙**name lps indicates longest proper prefix which is also suffix.**. A proper prefix is prefix with whole string **not**allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC” and “ABC”.

∙We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.

∙For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

**BOYER MOORE ALGORITHM:**

Boyer Moore is a combination of following two approaches.

1)Bad Character Heuristic

2)Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text.It processes the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by the max of the slides suggested by the two heuristics. **So it uses best of the two heuristics at every step**.

Unlike the previous pattern searching algorithms, **Boyer Moore algorithm starts matching from the last character of the pattern.**

**Bad Character Heuristic**

The idea of bad character heuristic is simple. The character of the text which doesn’t match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

1)The mismatch becomes a match

2)Pattern P move past the mismatched character.

**Case 1 – Mismatch become match**

We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we’ll shift the pattern such that it get aligned to the mismatching character in text T.



**Explanation:** In the above example, we got a mismatch at position 3. Here our mismatching character is “A”. Now we will search for last occurrence of “A” in pattern. We got “A” at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that “A” in pattern get aligned with “A” in text.

**Case 2 – Pattern move past the mismatch character** We’ll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



In the following implementation, we preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

**ROBIN KARP ALGORITHM:**

Robin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself. 2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property. 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+m-1]) 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.

The hash function suggested by Rabin and Karp calculates an integer value. The integer valuefor a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of “122” will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

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

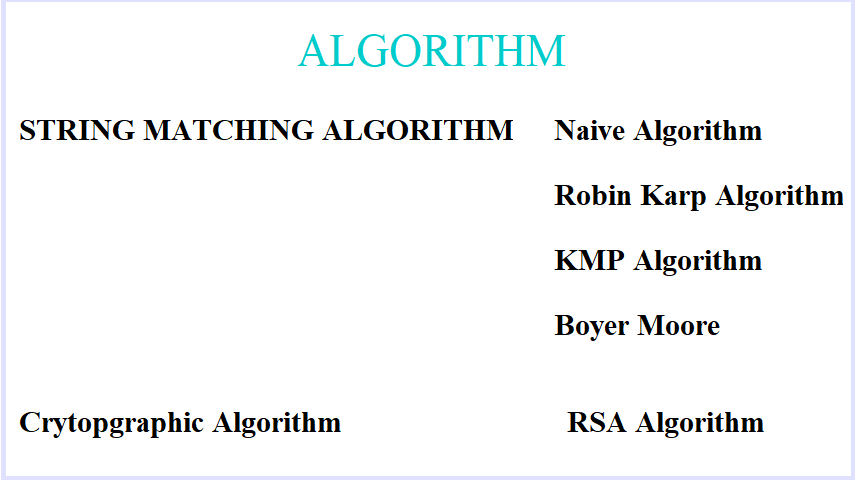
hash( txt[s .. s+m-1] ) : Hash value at shift s. hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1) d: Number of characters in the alphabet q: A prime number h: d^(m-1)

The average and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[].

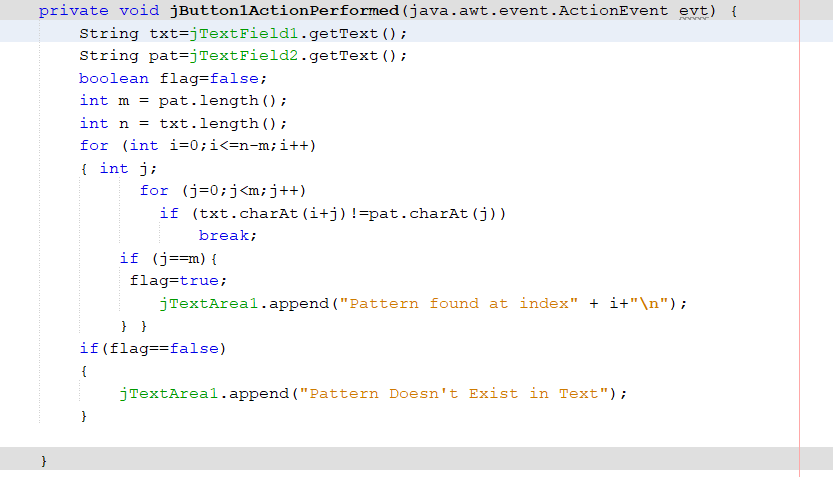
***IMPLEMENTATION:-***

**The Program are run on Java using software Netbeans which Provides a Easy User Interaction:-**

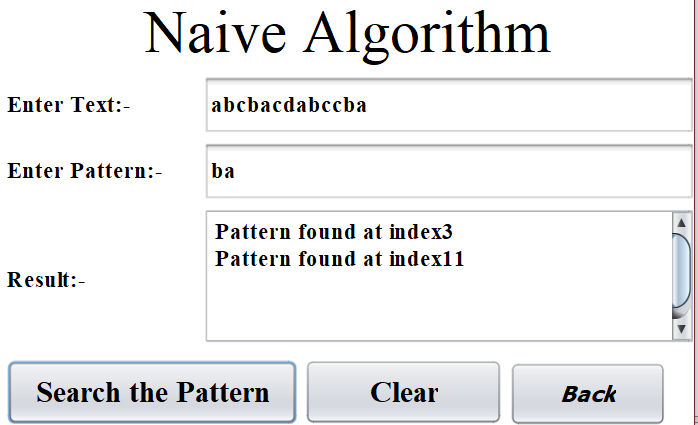
**Below is the Programs And their codes with Example:-**

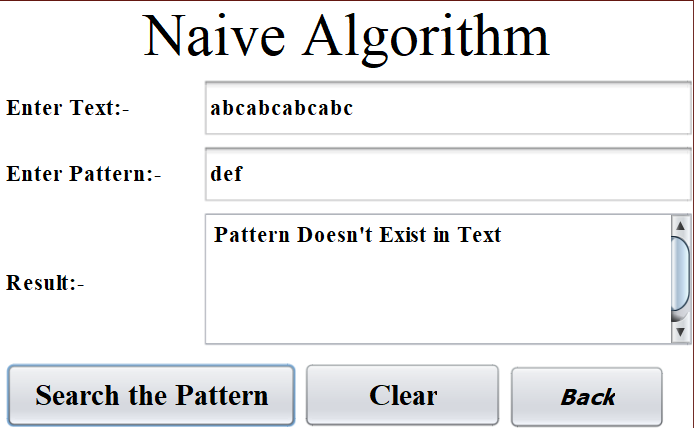
**

***Naïve:-***

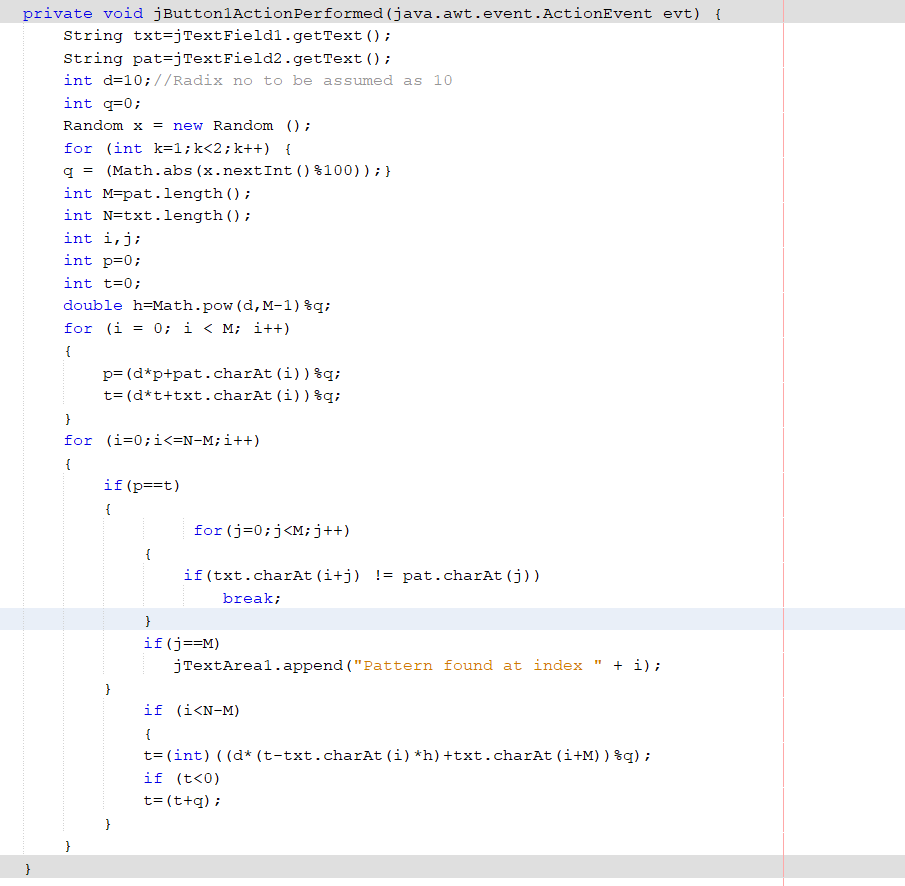
******

***Example:-*** Case :- When String Contains the Pattern and when it doesn’t

******

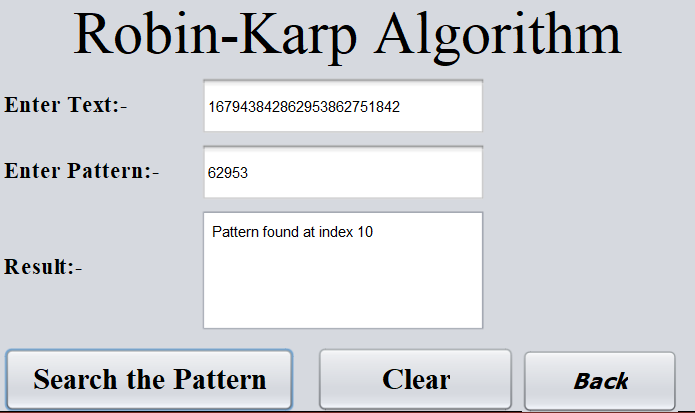
******

***Robin Karp:-***

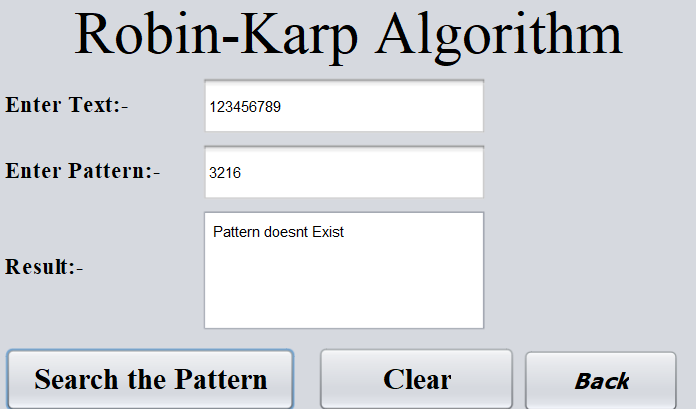


***Example:-***

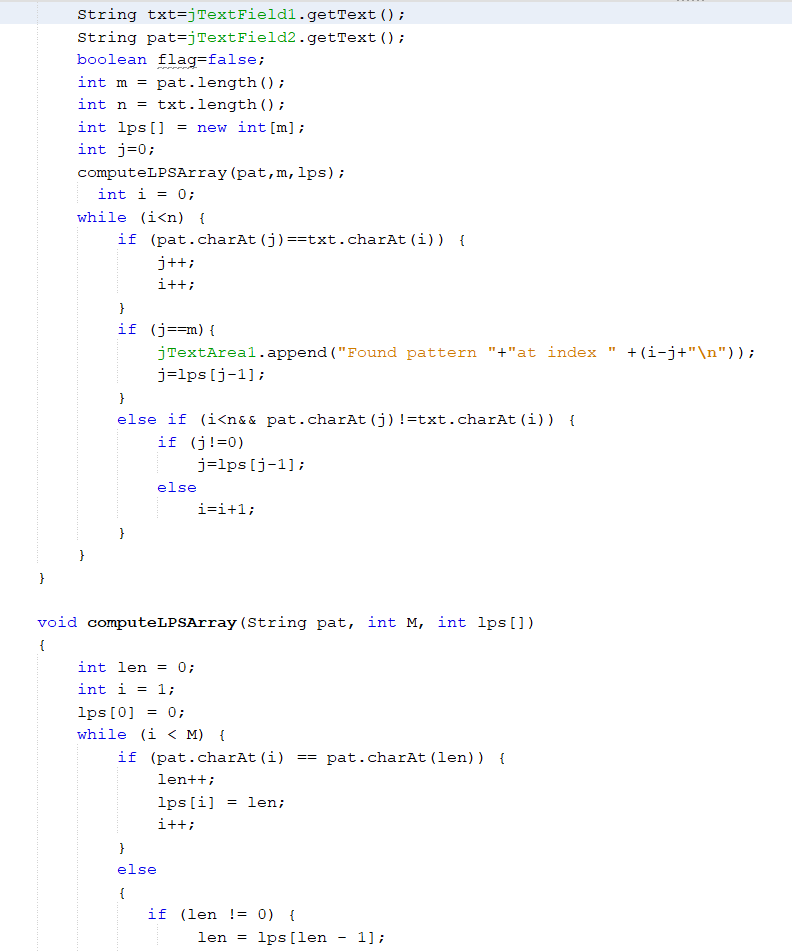
Case :- When String Contains the Pattern

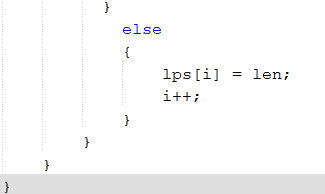


Case :- When String Doesn’t Contains the Pattern



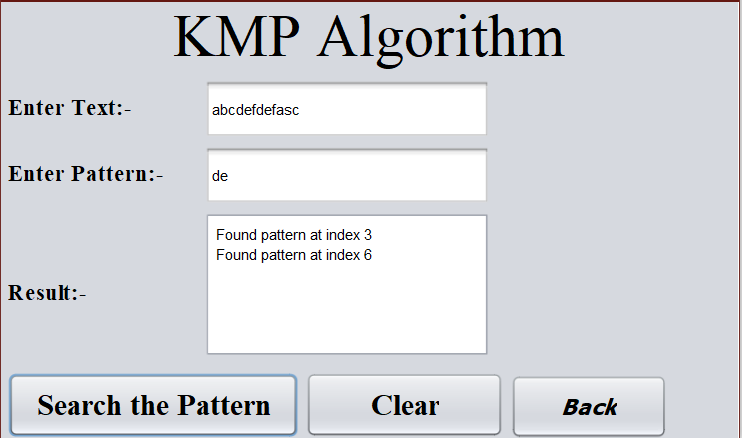
***KMP Algorithm:-***



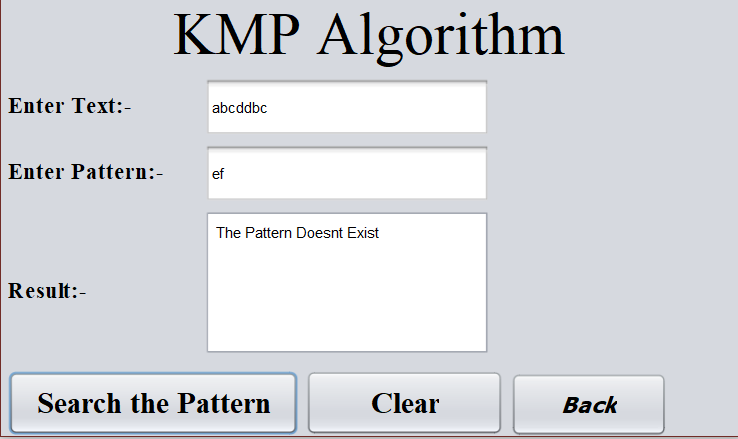


Examples:-

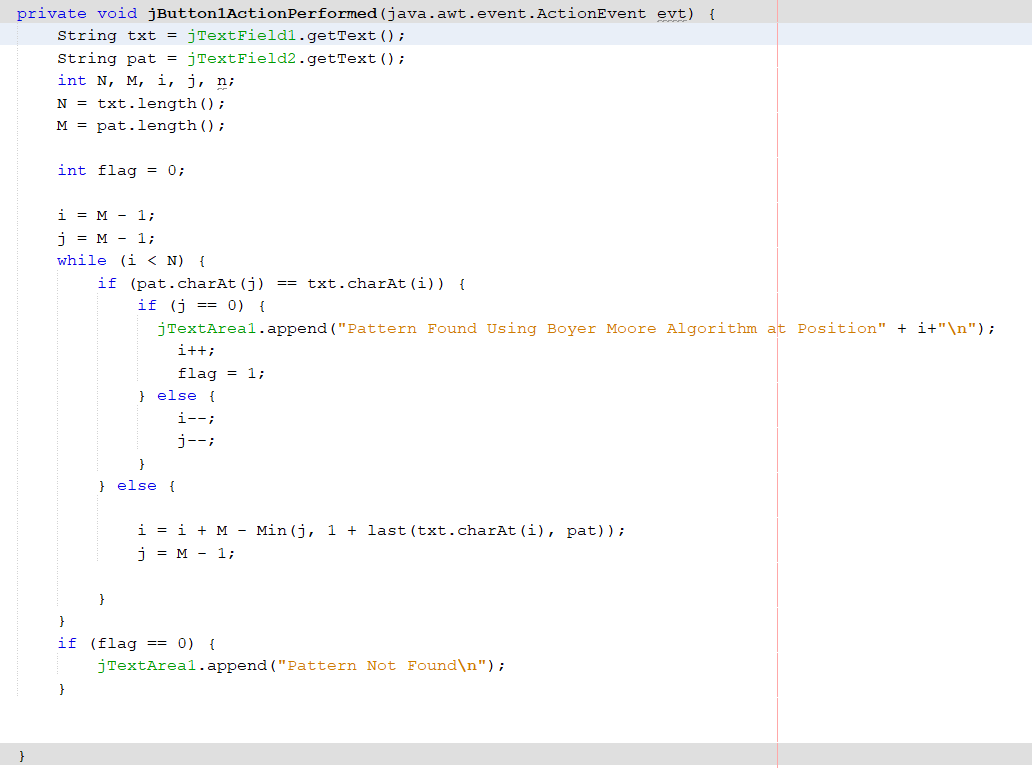
Case :- When String Contains the Pattern

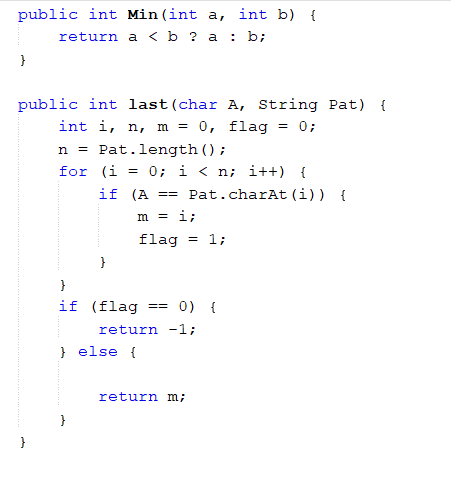


Case :- When String Doesn’t Contains the Pattern



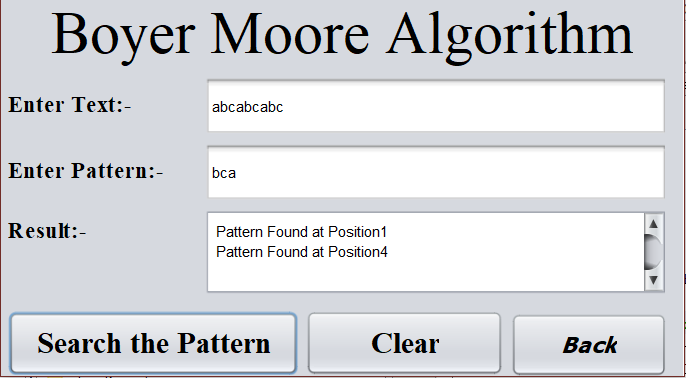
***Boyer Moore:-***



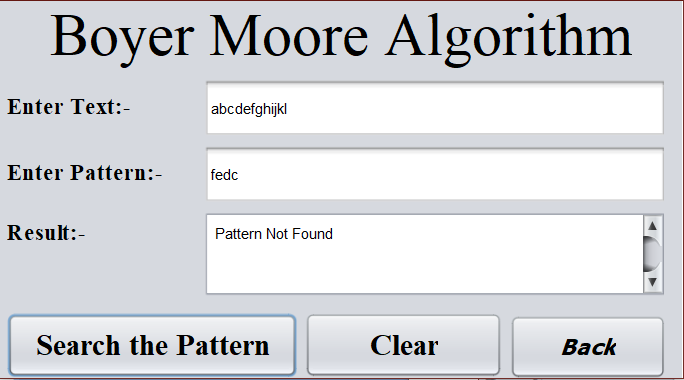


Examples:-

Case :- When String Contains the Pattern



Case :- When String Doesn’t Contains the Pattern



***Cryptographic Algorithm :-***

***Rsa Algorithm:-***

