AISSMS
COLLEGE OF ENGINEERING
ज्ञानम् सकलजनहिताय
Accredited by NAAC with "A+" Grade

Department Of Computer Engineering

2022-2023

"Implement the Naive string-matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input."

Submitted to the

Savitribai Phule Pune University

In partial fulfilment for the award of the Degree of

Bachelor of Engineering

in

Computer Engineering

By

1)	Kshitij Bhilare.	19CO009
2)	Atharva Jagtap.	19CO031
3)	Mahant Wagh.	19CO071
4)	Abhilash Yadnik.	19CO072

Under the guidance of

Prof. N.R.Talhar.

AISSMS
COLLEGE OF ENGINEERING
ज्ञानम् सकलजनहिताय
Accredited by NAAC with "A+" Grade

CERTIFICATE

This is to certify that the mini project report entitled **"Implement the Naive string-matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input."**
being submitted **Kshitij Bhilare, Atharva Jagtap, Mahant Wagh, Abhilash Yadnik** is a record of bonafide work carried out by him/her under the supervision and guidance of
Prof. N.R.Talhar in partial fulfilment of the requirement for **BE (Computer Engineering) – 2019 course** of Savitribai Phule Pune University, Pune in the academic year 2022-2023.

Date:

Place: Pune

Subject Coordinator   Head of the Department

Principal

This Mini Project report has been examined by us as per the Savitribai Phule Pune University, Pune requirements at **AISSMS COLLEGE OF ENGINEERING** Pune – 411001 on . . . .

Internal Examiner   External Examiner

ACKNOWLEDGEMENT

First and foremost, praises and thanks to the God, the Almighty, for showers of blessings throughout my project work to complete the research successfully.

I would like to express my deep and sincere gratitude to my subject teacher **Prof. N.R.Talhar** for giving us the opportunity to do this project and provide invaluable guidance throughout this project. Her dynamism, vision, sincerity and motivation have deeply inspired us. She has taught us the methodology to carry out the research and to present the project works as clearly as possible. It was a great privilege and honour to work and study under her guidance. We are extremely grateful for what she has offered us. We would also like to thank her for her friendship, empathy, and great sense of humour.

We are extremely grateful to all group members **Kshitij Bhilare, Atharva Jagtap, Mahant Wagh, Abhilash Yadnik** for the dedication and consistency towards this mini project. And also thankful for all the resources which are provided by each group member and which played a very crucial rolein the accomplishment of this project.

CONTENTS

Abstract

In every text-editing application like notepad, MS Word, MS Excel, etc. There is a need of mechanism to find a particular pattern. The main objective of string or pattern searching is to search particular pattern for position in a large body of text (e.g. From a book, a paragraph, a sentence, etc.). The purpose is to find occurrence of a text within another text. For example, when we need to find some text in text editor it is a tough task to find that word or text manually. To make our task simple we simply press shortcut key Ctrl+F, which generates a pop-up window where we can enter text which we need to search and it finds the occurrences of that particular text in the entire document. If matches found then it will highlight it all the occurrence of string which we are looking for otherwise it will display no matches found if there is zero occurrence of string. This is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.

Introduction

String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems. It helps in performing time-efficient tasks in multiple domains. These algorithms are useful in the case of searching a string within another string. String matching is also used in the Database schema, Network systems. Let us look at a few string-matching algorithms before proceeding to their applications in real world. String Matching Algorithms can broadly be classified into two types of algorithms –

- Exact String-Matching Algorithms
- Approximate String-Matching Algorithms

Exact String-Matching Algorithms:
Exact string-matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence. These are further classified into four categories:

Algorithms based on character comparison:

- Naive Algorithm: It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
- Using the Trie data structure: It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST.

Deterministic Finite Automaton (DFA) method:

Automaton Matcher Algorithm: It starts from the first state of the automata and the first character of the text. At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state.

Hashing-string matching algorithms:

Rabin Karp Algorithm: It 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.

Problem Statement

Implement the Naive string-matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.

Motivation

The motivation behind this paper is to study about various algorithms for the String-matching issue. These algorithms are utilized for attempting to find one, a few or all events of a characterized pattern in a larger text. The string-matching problem has various applications in different regions. Initially, an adjusted and productive algorithm of this issue can help to upgrade the responsiveness of a text editing program. Different applications in information technology incorporates web search engines, spam filters, natural language processing, computational science (enquiry of specific example in DNA arrangement), and feature detection in digital images.

Objectives

1) To understand and explore the working of naïve string algorithm for string matching.

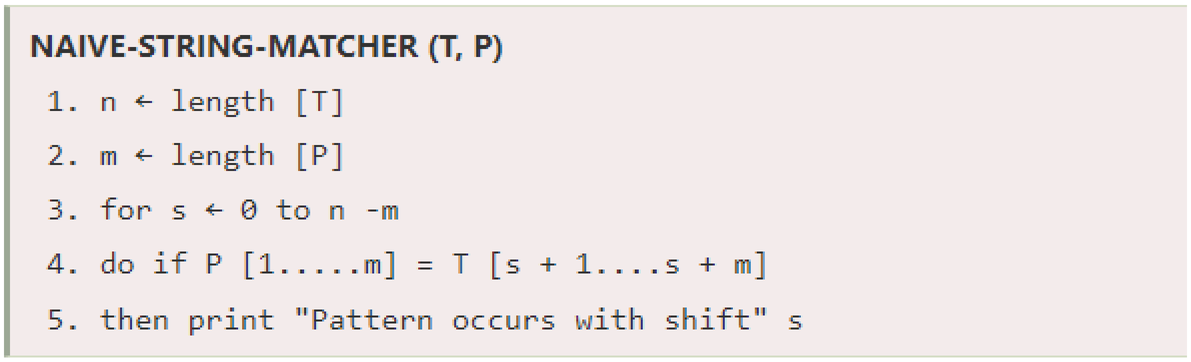
2) To understand and explore the working of Rabin-Karp algorithm for string-matching.

Theory

Naïve String-Matching Algorithm:

The naïve approach tests all the possible placement of Pattern P [1. ......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1..... m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1...... s+m] for each of the n - m +1 possible value of s.



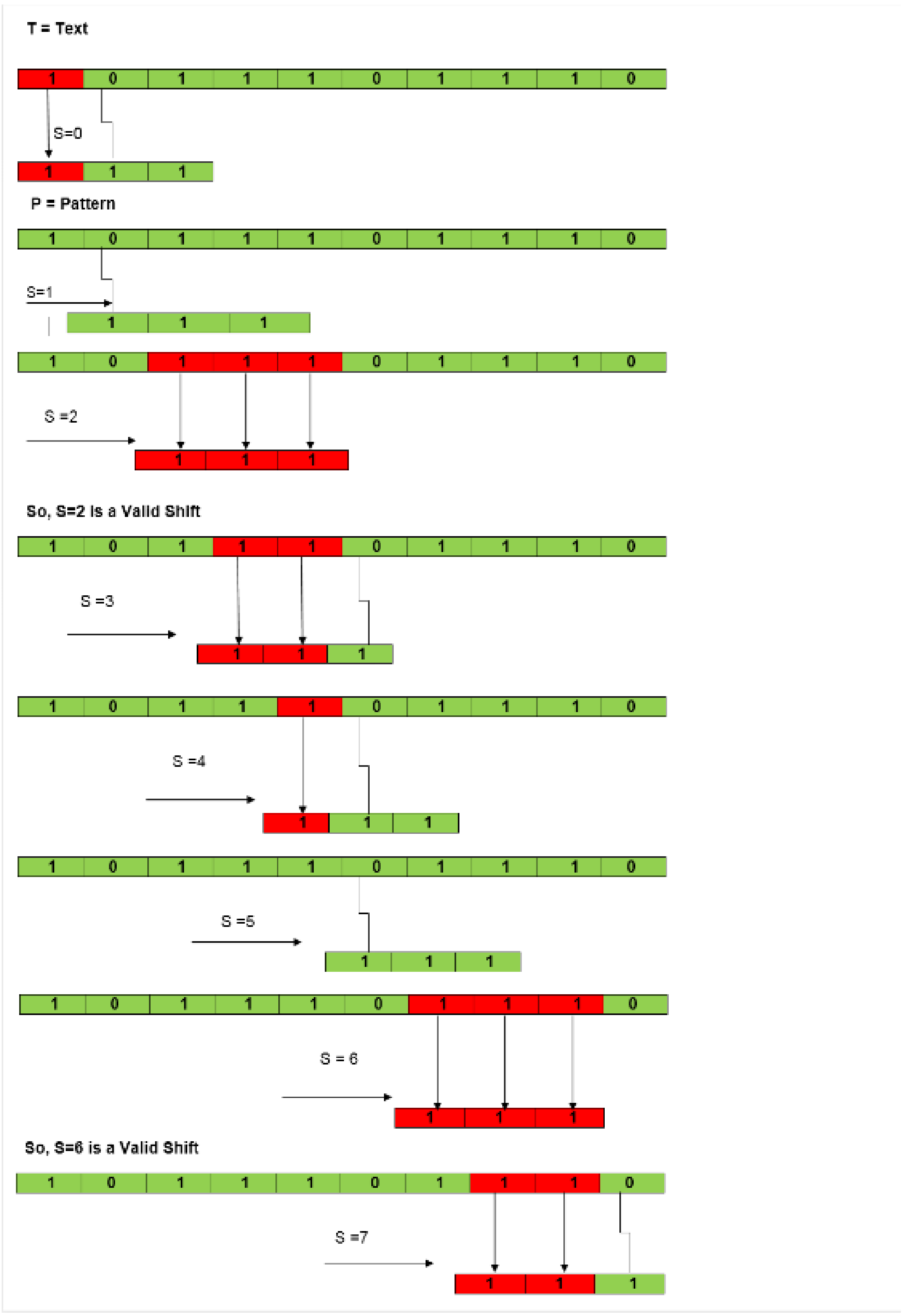
NAIVE-STRING-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

Example:

1. Suppose T = 1011101110
2. P = 111
3. Find all the Valid Shift

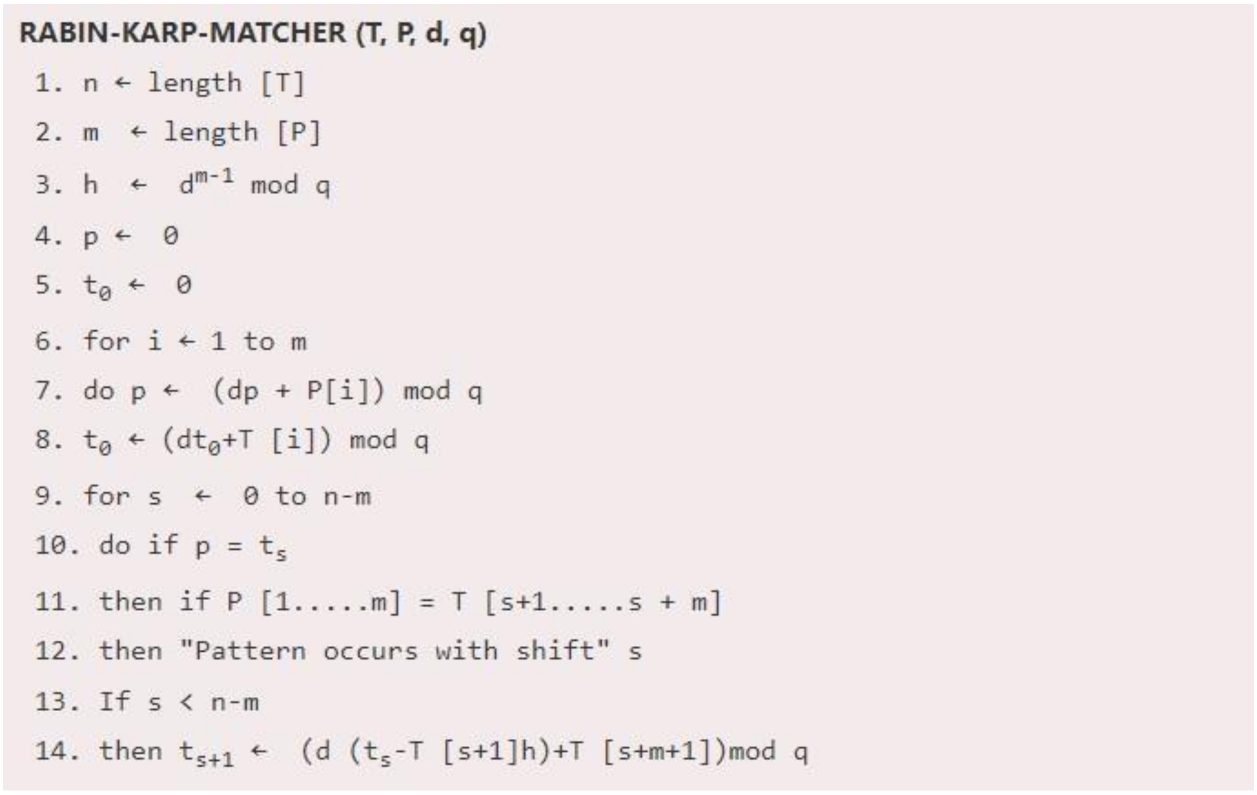
Solution:



T = Text
1 0 1 1 1 0 1 1 1 0
S=0
1 1 1
P = Pattern
S=1
S =2
So, S=2 is a Valid Shift
S =3
S =4
S =5
S = 6
So, S=6 is a Valid Shift
S =7

<u>Rabin-Karp String-Matching Algorithm:</u>

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.



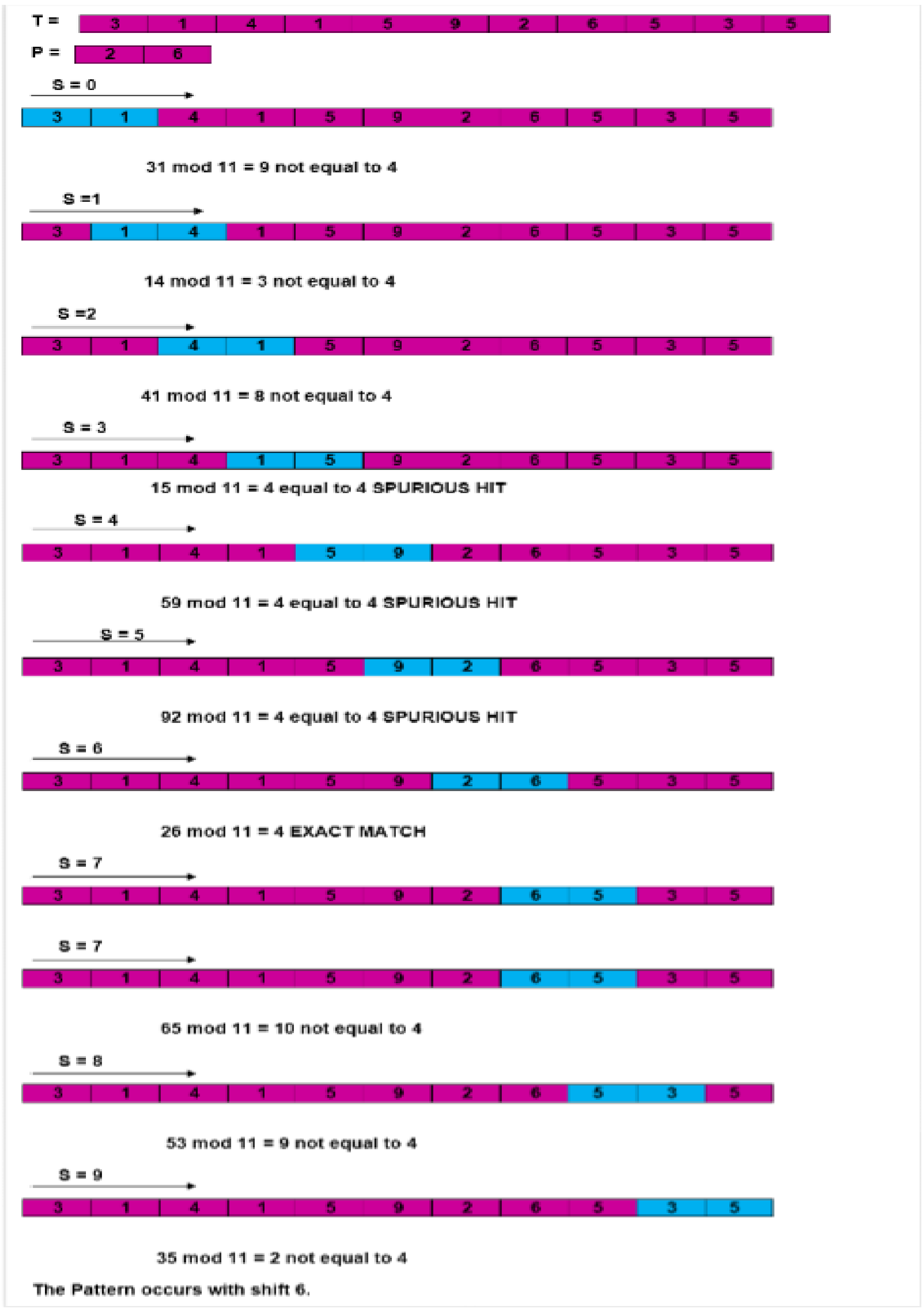
RABIN-KARP-MATCHER (T, P, d, q)

```
n ← length [T]
m ← length [P]
h ← $d^{m-1}$ mod q
p ← 0
$t_0$ ← 0
for i ← 1 to m
do p ← (dp + P[i]) mod q
$t_0$ ← ($dt_0$+T [i]) mod q
for s ← 0 to n-m
do if p = $t_s$
then if P [1.....m] = T [s+1.....s + m]
then "Pattern occurs with shift" s
If s < n-m
then $t_{s+1}$ ← (d ($t_s$-T [s+1]h)+T [s+m+1])mod q
```

Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.......

1. T = 31415926535.......
2. P = 26
3. Here T, Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

Solution:


T = 3 1 4 1 5 9 2 6 5 3 5
P = 2 6
S = 0
31 mod 11 = 9 not equal to 4
S =1
14 mod 11 = 3 not equal to 4
S =2
41 mod 11 = 8 not equal to 4
S = 3
15 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 4
59 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 5
92 mod 11 = 4 equal to 4 SPURIOUS HIT
S = 6
26 mod 11 = 4 EXACT MATCH
S = 7
S = 7
65 mod 11 = 10 not equal to 4
S = 8
53 mod 11 = 9 not equal to 4
S = 9
35 mod 11 = 2 not equal to 4
The Pattern occurs with shift 6.

Implementation of Naïve String-Matching Algorithm:

Input:

Main String: “ABAAABCDBBABCDDEBCABC”, pattern: “ABC”

Output:

Pattern found at position: 4

Pattern found at position: 10

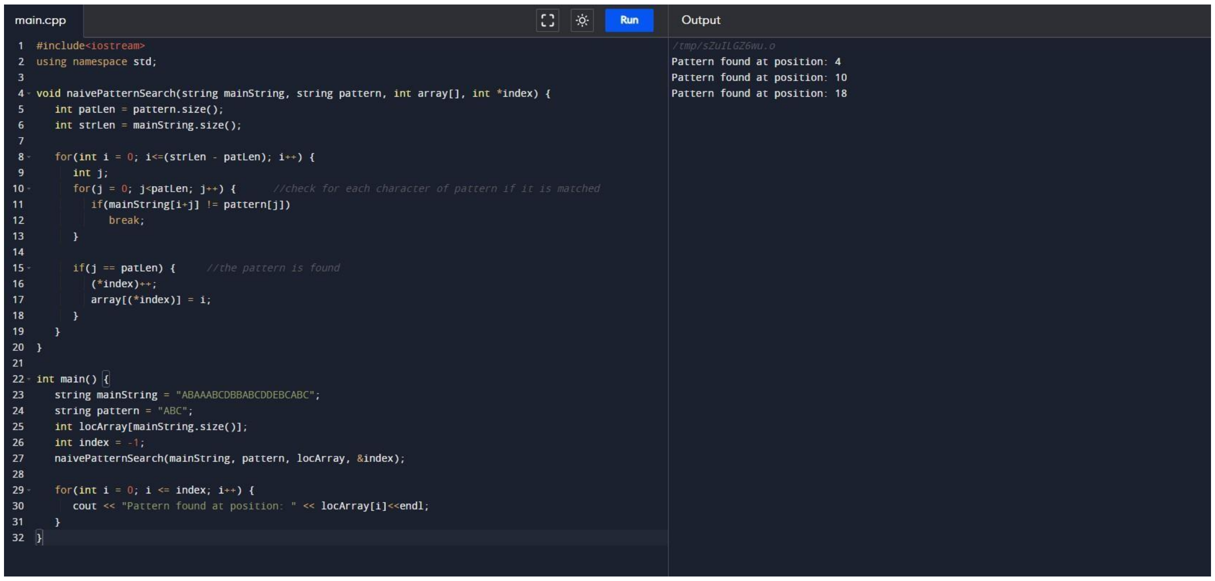
Pattern found at position: 18

Analysis: The time complexity of Naïve Pattern Search method is O(m*n). The m is the size of pattern and n is the size of the main string.

Code:

```
#include<iostream>
using namespace std;

void naivePatternSearch(string mainString, string pattern, int array[], int *index) {
  int patLen = pattern.size();
  int strLen = mainString.size();

  for(int i = 0; i<=(strLen - patLen); i++) {
    int j;
    for(j = 0; j<patLen; j++) {      //check for each character of pattern if it is matched
      if(mainString[i+j] != pattern[j])
        break;
    }

    if(j == patLen) {     //the pattern is found
      (*index)++;
      array[(*index)] = i;
    }
  }
}

int main() {
  string mainString = "ABAAABCDBBABCDDEBCABC";
  string pattern = "ABC";
  int locArray[mainString.size()];
  int index = -1;
  naivePatternSearch(mainString, pattern, locArray, &index);

  for(int i = 0; i <= index; i++) {
    cout << "Pattern found at position: " << locArray[i]<<endl;
  }
}
```

OUTPUT:

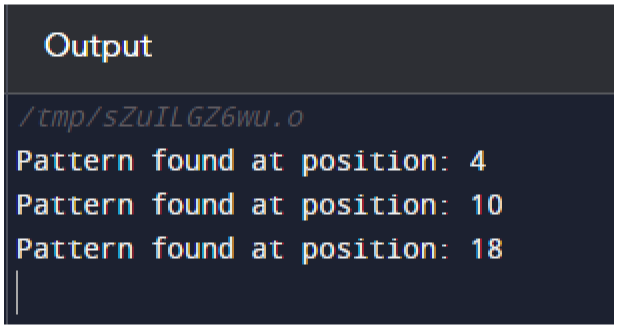


main.cpp

```
#include<iostream>
using namespace std;

void naivePatternSearch(string mainString, string pattern, int array[], int *index) {
   int patLen = pattern.size();
   int strLen = mainString.size();

   for(int i = 0; i<=(strLen - patLen); i++) {
      int j;
      for(j = 0; j<patLen; j++) {      //check for each character of pattern if it is matched
         if(mainString[i+j] != pattern[j])
            break;
      }

      if(j == patLen) {     //the pattern is found
         (*index)++;
         array[(*index)] = i;
      }
   }
}

int main() {
   string mainString = "ABAAABCDBBABCDDEBCABC";
   string pattern = "ABC";
   int locArray[mainString.size()];
   int index = -1;
   naivePatternSearch(mainString, pattern, locArray, &index);

   for(int i = 0; i <= index; i++) {
      cout << "Pattern found at position: " << locArray[i]<<endl;
   }
}
```

Output

```
/tmp/sZuILGZ6mu.o
Pattern found at position: 4
Pattern found at position: 10
Pattern found at position: 18
```



```
Output
/tmp/sZuILGZ6wu.o
Pattern found at position: 4
Pattern found at position: 10
Pattern found at position: 18
```

Implementation of Rabin-Karp String-Matching Algorithm:

Input:

Main String: “ABAAABCDBBABCDDEBCABC”, Pattern “ABC”

Output:

Pattern found at position: 4

Pattern found at position: 10

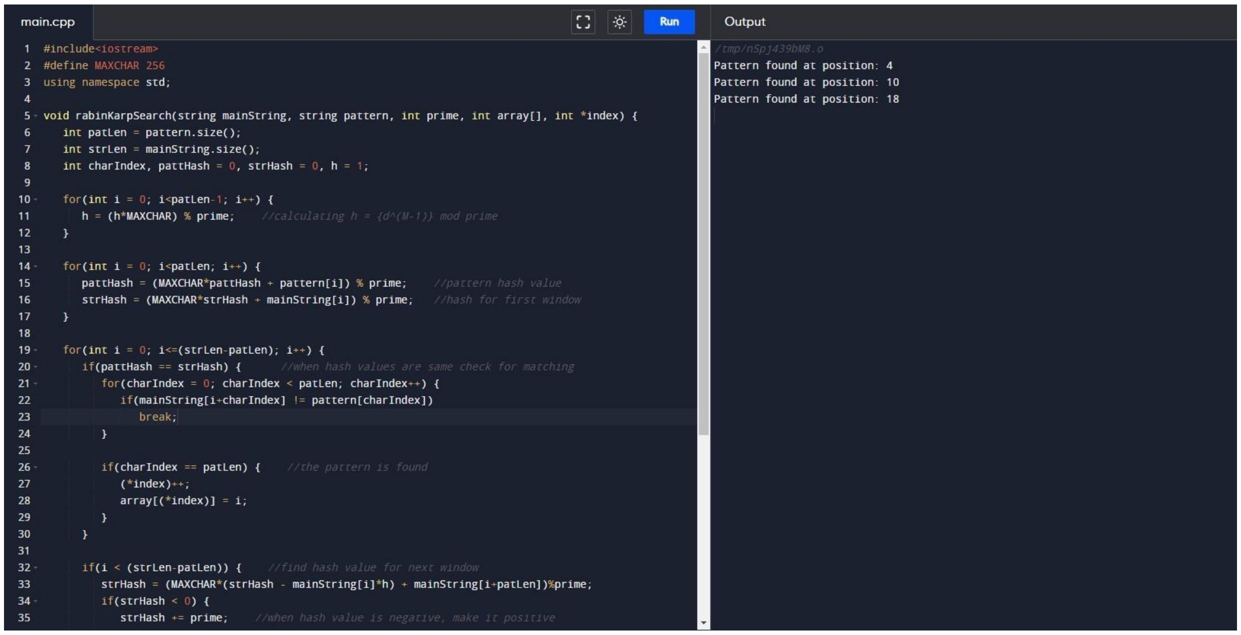
Pattern found at position: 18

Analysis: The time complexity is O(m+n), but for the worst case, it is O(mn).

Code:

```
#include<iostream>
#define MAXCHAR 256
using namespace std;

void rabinKarpSearch(string mainString, string pattern, int prime, int array[], int *index) {
  int patLen = pattern.size();
  int strLen = mainString.size();
  int charIndex, pattHash = 0, strHash = 0, h = 1;

  for(int i = 0; i<patLen-1; i++) {
    h = (h*MAXCHAR) % prime;    //calculating h = {d^(M-1)} mod prime
  }

  for(int i = 0; i<patLen; i++) {
    pattHash = (MAXCHAR*pattHash + pattern[i]) % prime;    //pattern hash value
    strHash = (MAXCHAR*strHash + mainString[i]) % prime;  //hash for first window
  }

  for(int i = 0; i<=(strLen-patLen); i++) {
    if(pattHash == strHash) {     //when hash values are same check for matching
      for(charIndex = 0; charIndex < patLen; charIndex++) {
        if(mainString[i+charIndex] != pattern[charIndex])
          break;
      }

      if(charIndex == patLen) {    //the pattern is found
        (*index)++;
        array[(*index)] = i;
      }
    }

    if(i < (strLen-patLen)) {    //find hash value for next window
      strHash = (MAXCHAR*(strHash - mainString[i]*h) + mainString[i+patLen])%prime;
      if(strHash < 0) {
        strHash += prime;    //when hash value is negative, make it positive
      }
    }
  }
}

int main() {
  string mainString = "ABAAABCDBBABCDDEBCABC";
  string pattern = "ABC";
  int locArray[mainString.size()];
  int prime = 101;
  int index = -1;
  rabinKarpSearch(mainString, pattern, prime, locArray, &index);

  for(int i = 0; i <= index; i++) {
    cout << "Pattern found at position: " << locArray[i]<<endl;
  }
}
```

OUTPUT:

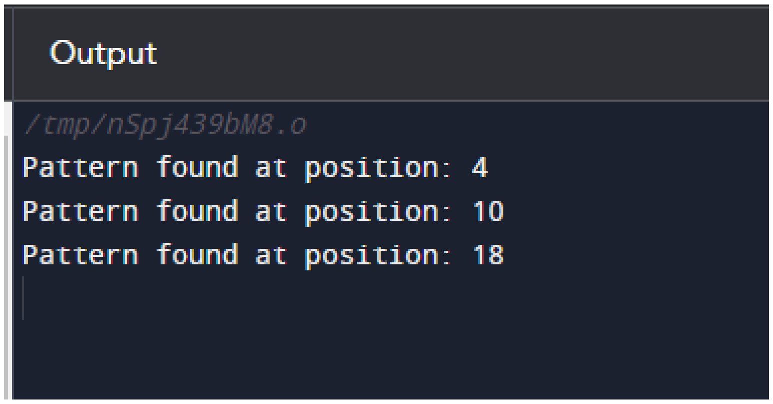


main.cpp

```
#include<iostream>
#define MAXCHAR 256
using namespace std;

void rabinKarpSearch(string mainString, string pattern, int prime, int array[], int *index) {
   int patLen = pattern.size();
   int strLen = mainString.size();
   int charIndex, pattHash = 0, strHash = 0, h = 1;

   for(int i = 0; i<patLen-1; i++) {
      h = (h*MAXCHAR) % prime;     //calculating h = {d^(M-1)} mod prime
   }

   for(int i = 0; i<patLen; i++) {
      pattHash = (MAXCHAR*pattHash + pattern[i]) % prime;     //pattern hash value
      strHash = (MAXCHAR*strHash + mainString[i]) % prime;    //hash for first window
   }

   for(int i = 0; i<=(strLen-patLen); i++) {
      if(pattHash == strHash) {       //when hash values are same check for matching
         for(charIndex = 0; charIndex < patLen; charIndex++) {
            if(mainString[i+charIndex] != pattern[charIndex])
               break;
         }

         if(charIndex == patLen) {     //the pattern is found
            (*index)++;
            array[(*index)] = i;
         }
      }

      if(i < (strLen-patLen)) {     //find hash value for next window
         strHash = (MAXCHAR*(strHash - mainString[i]*h) + mainString[i+patLen])%prime;
         if(strHash < 0) {
            strHash += prime;     //when hash value is negative, make it positive
```

Output

```
/tmp/nSpj439bM8.o
Pattern found at position: 4
Pattern found at position: 10
Pattern found at position: 18
```



Output

```
/tmp/nSpj439bM8.o
Pattern found at position: 4
Pattern found at position: 10
Pattern found at position: 18
```

Conclusion:

Hence, we have successfully implemented, understood the concept and working of naïve string-matching algorithm and Rabin-Karp string-matching algorithm in detail for same input.

References:

[1] https://www.tutorialspoint.com/index.htm

[2] https://www.javatpoint.com/

[3] https://www.geeksforgeeks.org/