****

**Advanced Algorithm design**

**GACS-7101-001/3**

**Rabin Karp Algorithm for string matching**

**Final Project Report**

**Winter 2018**

**Submitted to:- Prof. Yangjun Chen**

**Submitted by:- Hardeep Singh**

**Student no:- 3089450**

**Date:- April 16, 2018**

**Contents:**

1. **Introduction**
2. **Classification of string matching algorithm**
3. **Naïve Algorithm**
4. **Rabin-Karp Algorithm**
5. **Implementation**
6. **Code Diagrams**
7. **Output Diagrams**
8. **Comparison with other Algorithms**
9. **Conclusion**

**References**

**Introduction**

The main aim of the string matching algorithms is to find occurrences of a pattern within the given text. This occurrences can be for one or more time. When any string matching algorithm execute for the given data it returns the first array index where string matches.[algo for] There are various practical applications of the string matching in real life. Firstly, it can be used to increase the responsiveness of text-editing software. Other applications in computer science are spam filters, web search engines, natural language processing and DNA matching.

In this report first we go with the naïve string matching algorithm and will calculates its time complexity. After that, an algorithm with better time complexity will be discussed called Rabin-Karp algorithm.

**Formal definition of string matching:-**

Given Pattern = P[m] where *m* is the length of the pattern.

Text = T[n] where *n* is the length of the text.

In string matching problems *m* should always be lesser than *n* than only it leads to occurrence of pattern inside text.

**Example:-**

Given **Text** = Data structures and Databases

**Pattern** = Data

String is matching at array index 0 and 20.

**Classifications of string matching Algorithms:-**

1. **Classifications on the basis of number of patterns**

* Single pattern algorithms
* Naïve or Brute force
* Boyre-Moore
* Knuth-Morris-Pratt
* Rabin-Karp
* Finite set of patterns
* Aho-Corasick
* Rabin-Karp
* Commentz-Walter
* Set-BOM
* Infinite set of patterns
* Grammar checking

1. **Classification on the basis of matching strategy**

* Prefix matching strategy
* Aho-Corasick
* Knuth-Morris-Pratt
* Suffix matching strategy
* Commentz-Walter
* Boyre-Moore
* Best Factor match
* BNDM
* BOM
* Set BOM
* Other strategies
* Naïve
* Rabin-Karp

**Naïve string matching algorithm:**

A naïve string matching algorithm is easy but very inefficient method of finding pattern inside the given text. At first, it checks if there is copy of the needle in the first character of the text. If it doesn’t match than the needle gets increment by 1 and finds if second character is matching or not. Then it checks for third character and continues till the end of the given text. Moreover, as it is a straight forward algorithm there is zero pre-processing time. The processing directly starts from the matching of the pattern with the text, and if characters matches it returns the index value.

**Formal Algorithm :**

* Initialize n as the length of the given text.
* Initialize m as the length of the entered pattern.
* For i=0 to n-m
* If Pattern[m] == Text[i]
* Return index where string matches.

The following algorithm shows that pattern window slides 1 at a time. If characters of pattern matches with the text than it returns the index value.

**Time complexity of Naïve algorithm:**

The following given time complexity is for naïve algorithm. As naïve algorithm dosen’t have any pre-processing time still its matching time is coming out to be O(MN), which is inefficient in the real time working environment. This leads to development of various other string matching algorithms.

Best case:

O(M)

Worst Case:

O(MN)

**Other approach:**

The main concern of the string matching algorithms is time complexity. There is always requirement of the faster and efficient algorithm for string matching to reduce the overall cost. The naïve algorithm is matching characters of the pattern and the text, which is consuming more time.

We can skip this character comparisons by introduction of the hash function. Hash function used to calculate the hash value of the pattern and then compare it with the hash value of the substring of the text. When we go through all the string matching algorithms we finds that Rabin-Karp algorithm is best when it comes to the matching of hash values of the given pattern and text. In further section, we will discuss more about the Rabin-Karp algorithm and will see how hash function works.

**Definition for Rabin-Karp Algorithm:**

Rabin-Karp algorithm is a string matching algorithm which calculates the hash value of the pattern and then hash value of the substring from the text and compare it. It was created by Richard M.Karp and Michael O.Rabin in year 1987. Rabin-Karp algorithm compares the integer values instead of comparing the characters. After comparing for first set of string value window slide by one and then compares hash values for it again. Unlike naïve algorithm it also pre-process the string to calculate the hash value.

*Rolling Hash technique:-*

Rabin-Karp string matching uses rolling hash functions to calculate the hash values of the pattern and substring of text.

Formal equation:

Hash, H = c1xa-1 + c2xa-2 + c3xa-3 + …… + cax0

In the above equation c are the input characters and x is the constant.

There is one biggest advantage of the hash function that the calculated hash value is very small which is easy to carry in the overall algorithm. The small hash value comes because hash function is using the modulo (%) /prime number.

**Example of hash value calculations:**

Given substring: “bat”

Let the base be 63 and prime number be 113

ASCII of ‘b’, ‘a’ and ‘t’ are 98, 97 and 116 respectively

Applying hash function:-

98 x 632  + 97 x 631  + 116 x 630  = 395,189

Therefore,

395189%113 = 28

Hash value of “bat” would be 28.

**Rabin-Karp Algorithm:**

Assume *z* as base number and *y* asprime number*.*

1. Compute hashp for pattern P.
2. Compute hasht for the first substring of T with m length.
3. For i = 1 to n-m

If hashP = hashT

Match T[ i…. i+m] with P,

Else If matched then return 1.

Else hashT = (z(hashT – T[i+1].zm-1 +T[m+i+1]) mod y

1. End.

**Example using Rabin-Karp:**

Given Text,T = 31415926535 and Pattern,P = 26

Let Primeno, y= 11

P%y = 26%11 = 4 (hash value of pattern p)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

31%11 = 9, not equal to hash value of pattern 4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

14%11 = 3, not equal to hash value of pattern 4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

41%11 = 8, not equal to hash value of pattern 4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

15%11 = 4, equal to 4 spurious hit

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

59%11 = 4, equal to 4 spurious hit

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

92%11 = 4, equal to 4 spurious hit

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

26%11 = 4, EXACT MATCH

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

65%11 = 10, not equal to 4

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

53%11 = 9, not equal to 4

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

35%11 = 2, not equal to 4

When a match is found, still algorithm goes till the end of the text to ensure if there is another same pattern present or not.

**Implementation details:**

*Hardware Architecture:*

* Intel i5 processor.
* 4 GB RAM.

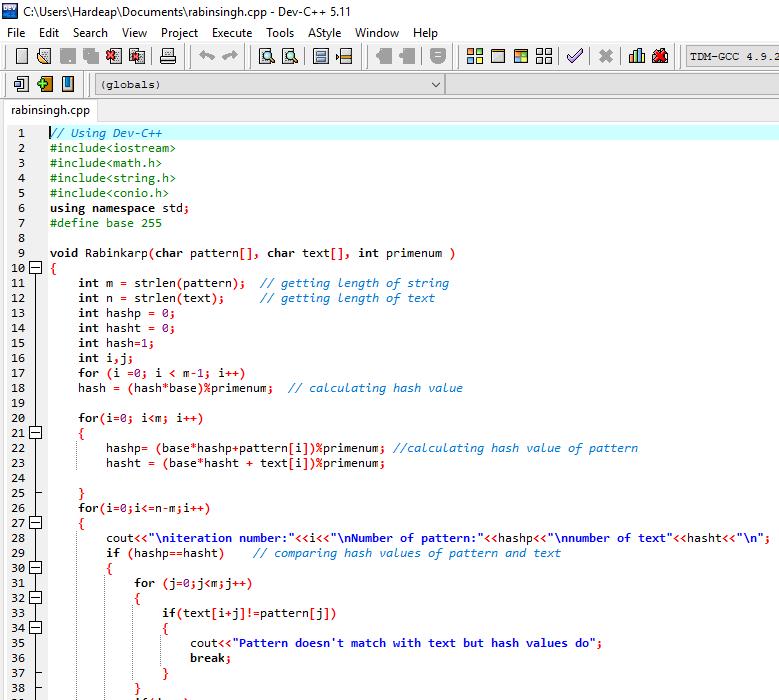
*Software Used:*

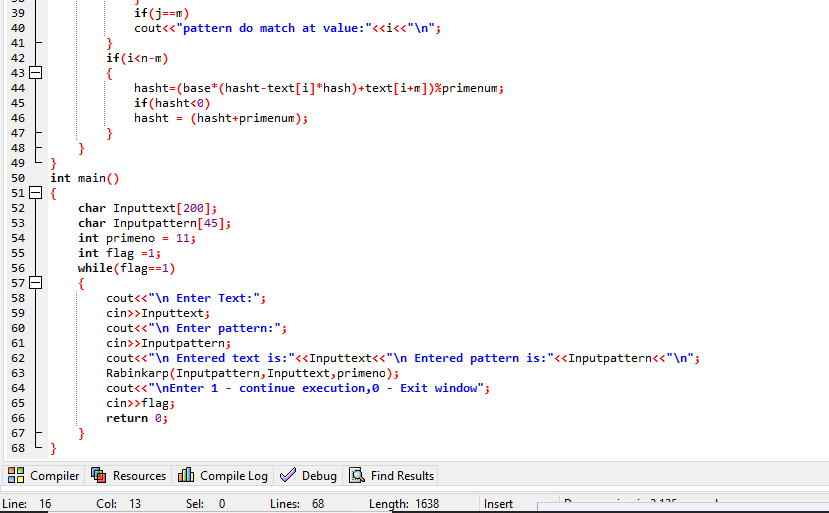
* DEV C++

*Language Used:*

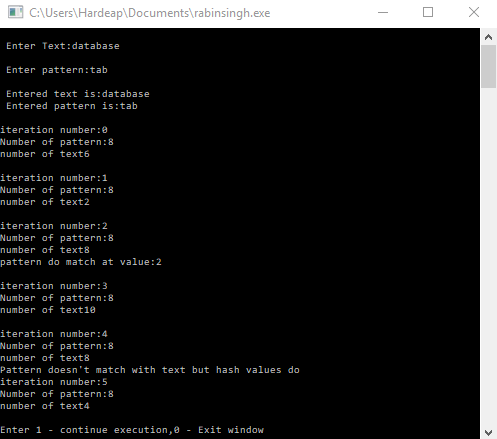
* C++

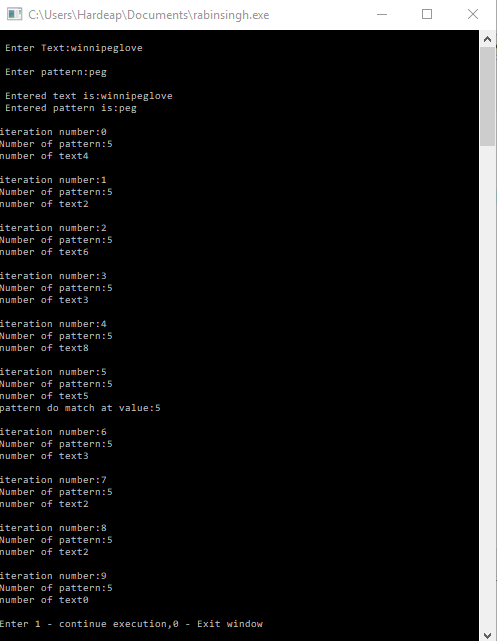
**Code diagrams:**

****



**Output diagrams:**

****

****

**Time complexity of Rabin-Karp Algorithm:**

Worst Case:

O(mn)

Avg Case:

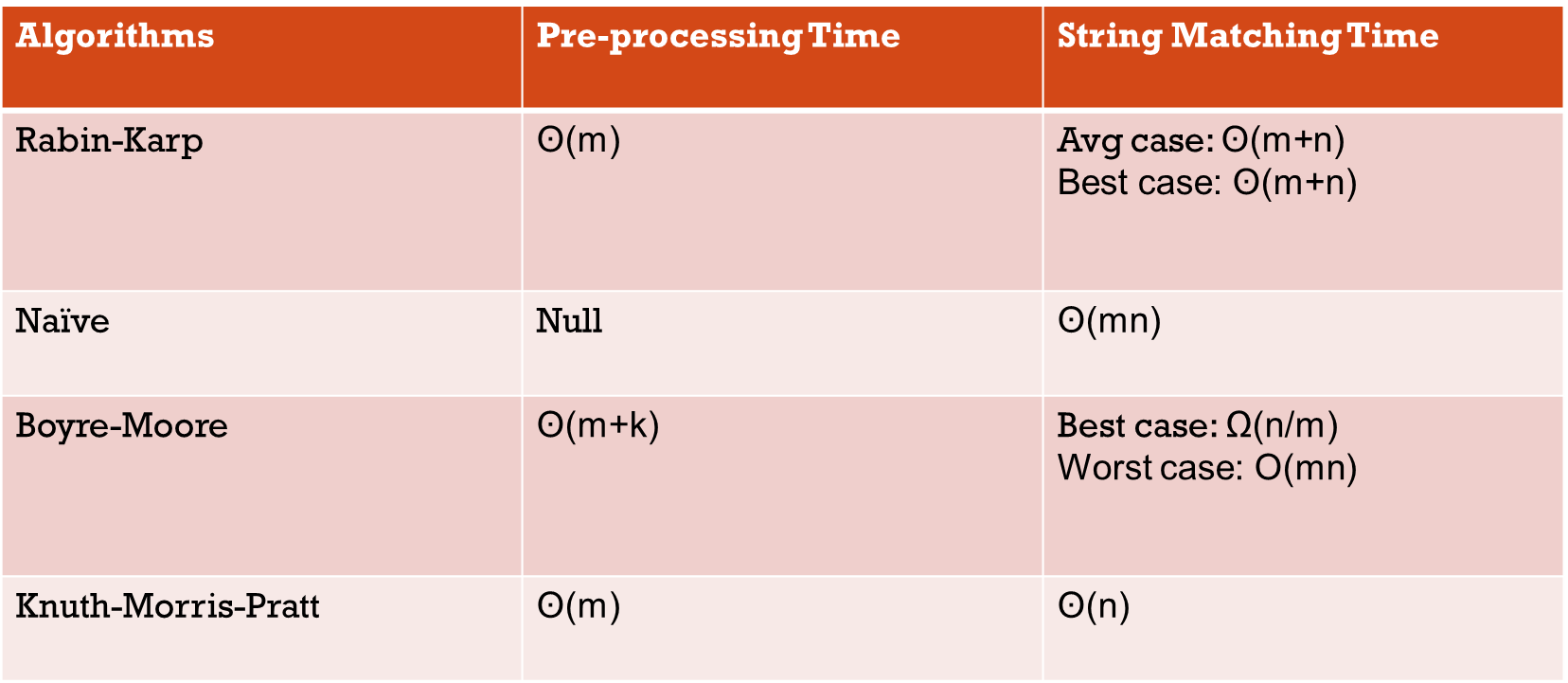
ʘ(m+n)

Best Case:

ʘ(m+n)

The best and average case of Rabin-Karp algorithm is efficient and fast whereas the worst case is slow because of the pre-processing time.

**Comparisons of Rabin-Karp with other string matching algorithms:**



**Applications of Rabin-Karp Algorithms:**

* **Bioinformatics**:-

It is used in looking for similarities of two or more proteins.

* **Search engines:-**

It is used to match the entered keyword in the search box with the files present on web.

* **Plagiarism detection:-**

It is used to detect plagiarism in academic work.

**Conclusion:**

The Rabin-Karp Algorithm is the efficient algorithm when it comes to best average case of string matching. Moreover, when it came to multiple pattern matching, it is best for that because it helps in plagiarism detection algorithms. However, it is inferior for single pattern matching to Knuth-Morris-Pratt, Boyer-Moore and various other single pattern search algorithms. The reason behind it is its slow worst case performance. If we want to search any large number of characters, let’s say x, a fixed length of patterns in a text, we can create a simple variant of the Rabin-Karp algorithm that uses a bloom filter or a set data structure to check whether hash values of patterns is matching with hash value of text or not.

**References:**

1. [**https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp\_algorithm**](https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm)
2. [**https://www.ijcsi.org/papers/IJCSI-9-1-1-219-226.pdf**](https://www.ijcsi.org/papers/IJCSI-9-1-1-219-226.pdf)
3. [**https://www.researchgate.net/profile/Andysah\_Putera\_Utama\_Siahaan/publication/319272358\_Examination\_of\_Document\_Similarity\_Using\_Rabin-Karp\_Algorithm/links/599fb0fc0f7e9b3639febb96/Examination-of-Document-Similarity-Using-Rabin-Karp-Algorithm.pdf**](https://www.researchgate.net/profile/Andysah_Putera_Utama_Siahaan/publication/319272358_Examination_of_Document_Similarity_Using_Rabin-Karp_Algorithm/links/599fb0fc0f7e9b3639febb96/Examination-of-Document-Similarity-Using-Rabin-Karp-Algorithm.pdf)
4. [**https://pdfs.semanticscholar.org/084e/cf9bccf38716c1d88e136a793aba169fdcd2.pdf**](https://pdfs.semanticscholar.org/084e/cf9bccf38716c1d88e136a793aba169fdcd2.pdf)