# C:\Users\user\Desktop\facebook_UL_profil_800x800px_5x_.png

UNIVERSITY OF LODZ

ADVANCED ALGORITHM

OGUZ ARAC E-5633

2020

# INDEX

PART ONE 3

1-Brute-Force 3

1.1-Example and Comparison 3

2-Sunday Algorithm 4

2.1-Example and Comparison 5

3-KMP Algorithm 6

3.1-Example and Comparison 6

4-FSM Algorithm 10

4.1-Example and Comparison 10

5-Rabin-Karp Algorithm 11

5.1-Example and Comparison 11

6-Conclusion 13

WACKY RACES 14

1-Sunday vs Brute-Force 14

2-Sunday vs KMP 14

3-KMP vs Rabin-Karp 15

4-Rabin-Karp vs Sunday 15

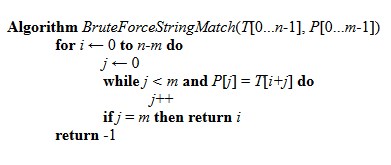
JEWISH-STYLE CARP 16

# PART ONE

## 1-Brute-Force

Brute-Force algorithm does not have shortcut to improve algorithm pace. It tries all possible way to find the given solution. Brute-Force working way checking, at all position in the text, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. The Brute-Force algorithm does not need extra space in addition to the pattern and the text, and it requires no preprocessing phase. The text character comparisons can be done in any order on the searching phase. The time complexity of the searching phase is O(nm).

Pseudo-Code:



### 1.1-Example and Comparison

Brute-Force algorithm useful for short text and pattern. It controls each character by shifting 1 step right. For instance,

Text: ”ABDHYSAUSW”

Pattern: “BDH”

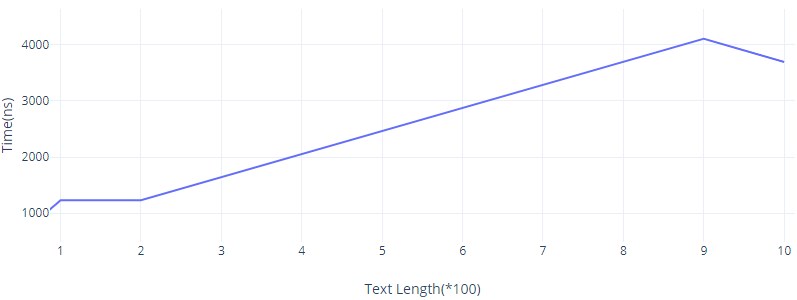
1. ABDHYSAUSW --Not matched

BDH

1. ABDHYSAUSW --Matched

BDH

Graph



n: Length of the text

m: Length of the pattern

* Worst case time complexity O(n.m).
* Best case time complexity if pattern found Ο(m).
* Best case time complexity if pattern not found: Ο(n).

What is the worst case?

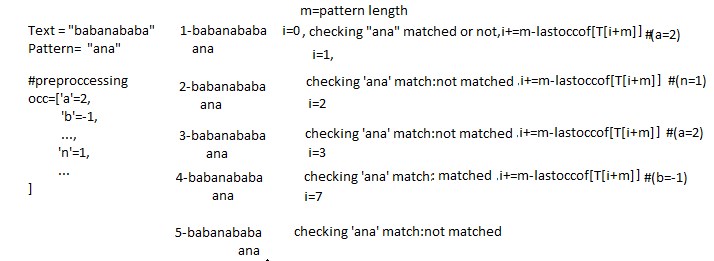
* When all characters of the text and pattern are same.
* Worst case also occurs when only the last character is different.

## 2-Sunday Algorithm

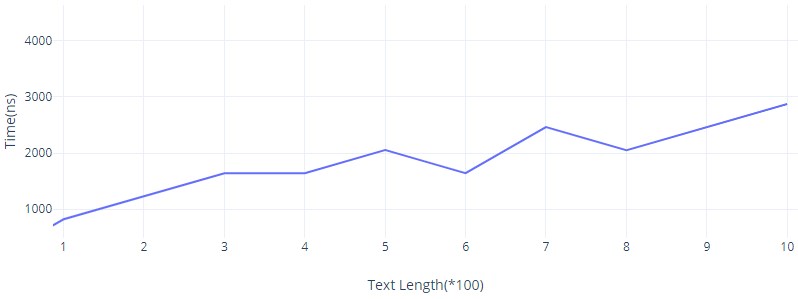
The Sunday algorithm has a preprocessing besides Brute-Force. In this process, The Sunday algorithm creates an array that is the length of ascii table and then fills all indexes with -1. After that, it changes the value in the table after checking characters of the pattern.

### 2.1-Example and Comparison

Example



Graph



* Algorithm performs just 0(n/m) comparison.
* As the length of the pattern gets longer, the algorithm execute faster.
* According to the time complexity, if pattern length is kept same, the algorithm need more time to execute.

## 3-KMP Algorithm

The KMP algorithm has a preprocessing phase which is created KMP table. Thanks to this table, KMP gets enough information to determine where to continue to search for non-correspondence.

The KMP algorithm is not re-examine the characters that were previously checked.

The time complexity of the preprocessing phase is O(m), The time complexity of the searching phase is O(m+n).

While the search is very fast in texts that do not usually contain many different characters, such as DNA, it is a bit slow in texts in the natural speech language style.

### 3.1-Example and Comparison

Example

Text: ‘ACAACACAACABACA’

Pattern: ‘ACABACA’

Preprocessing Phase,

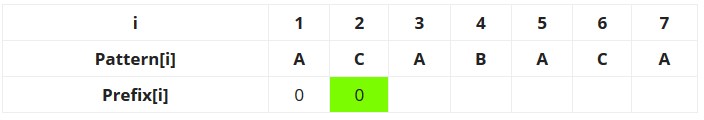
Step1

* Pattern: ACABACA
* i = 1 : A
* Prefix : none
* Suffix : none



Step2

* Pattern: ACABACA
* i = 2 : AC
* Prefix : A
* Suffix : C



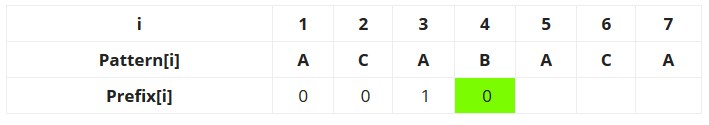
Step3

* Pattern: ACABACA
* i = 3 : ACA
* Prefix : A, AC
* Suffix : C, CA



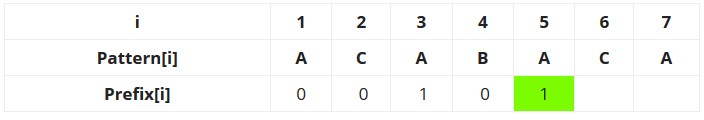
Step4

* Pattern: ACABACA
* i = 4 : ACAB
* Prefix : A, AC, ACA
* Suffix : C, CA, CAB



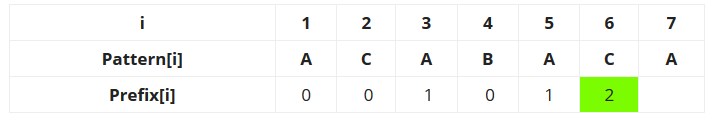
Step5

* Pattern: ACABACA
* i = 5 : ACABA
* Prefix : A, AC, ACA, ACAB
* Suffix : C, CA, CAB, CABA



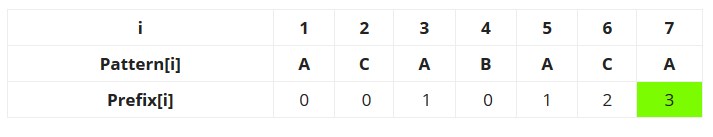
Step6

* Pattern: ACABACA
* i = 6 : ACABAC
* Prefix : A, AC, ACA, ACAB, ACABA
* Suffix : C, CA, CAB, CABA, CABAC

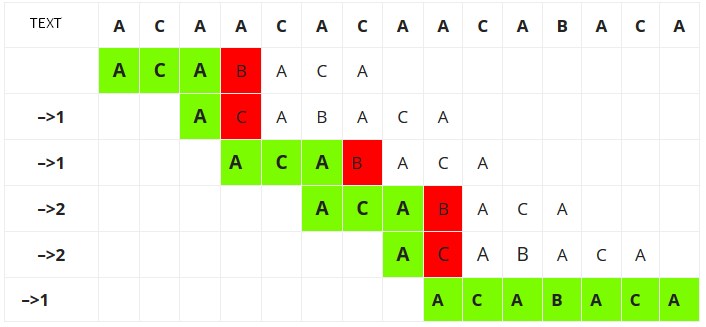


Step7

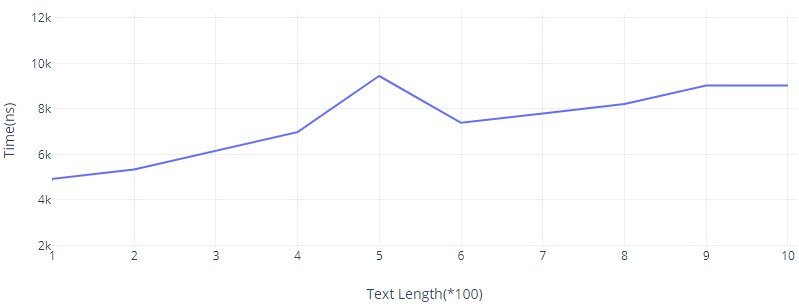
* Pattern: ACABACA
* i = 7 : ACABACA
* Prefix : A, AC, ACA, ACAB, ACABA, ACABCA
* Suffix : C, CA, CAB, CABA, CABAC, CABACA



Searching Phase



Graph



* The running time of Knuth-Morris-Pratt algorithm is proportional to the time needed to read the characters in text and pattern. The time complexity of the algorithm is O(m + n).
* It is important to note that these quantities are independent of the size of the underlying alphabet.
* Pattern length whatever it can be, running time always going to be O(m+n).

## 4-FSM Algorithm

FSM is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another.

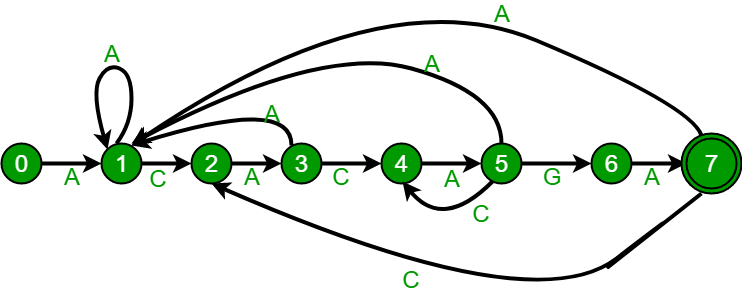
For the preprocessing, it uses a two-dimensional array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm.

In search, it starts from the first state and first character of the text.

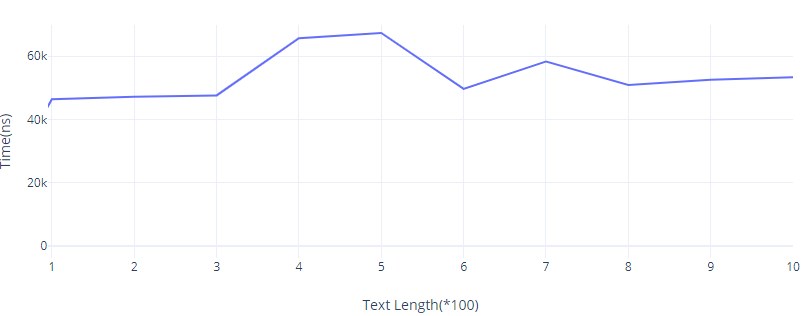
It considers next character of the text, looks for the next state and move to a new state. If it reaches to the final state, the pattern is found in the text.

### 4.1-Example and Comparison

A state diagram for the pattern ACACAGA.



Graph



* When the pattern length gets longer, FSM needs to more time to find match strings.

## 5-Rabin-Karp Algorithm

The Rabin-Karp Algorithm calculates a hash value for the pattern, and then it calculates a hash value for the m-character of the text. If the hash values are not equal, algorithm will calculate the hash value for next m-character. If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the m-character sequence.

### 5.1-Example and Comparison

Example

Text = "ABDAAAHDF"

Pattern = "AAAH"

hash\_p = 100 ('AAAH')

hash\_t = 46 ('ABDA')

1- ABDAAAHDF |

AAAH | 46 != 100, unmatched

hash\_t = 89 | hash value of next section of text, one character over ('BDAA')

2- ABDAAAHDF |

AAAH | 89 != 100, unmatched

hash\_t = 76 | hash value of next section. ('DAAA')

3- ABDAAAHDF |

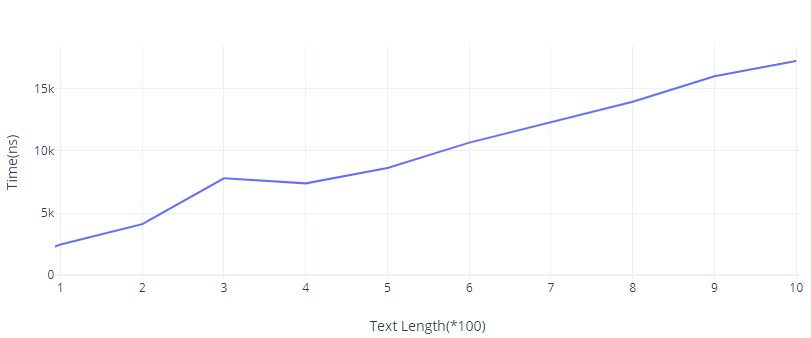
AAAH | 76 != 100, unmatched

hash\_t = 100 | hash value of next section. ('AAAH')

4- ABDAAAHDF |

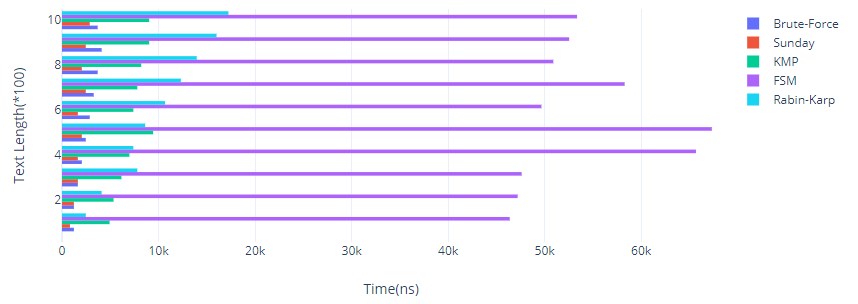
AAAH | 100 == 100, matched

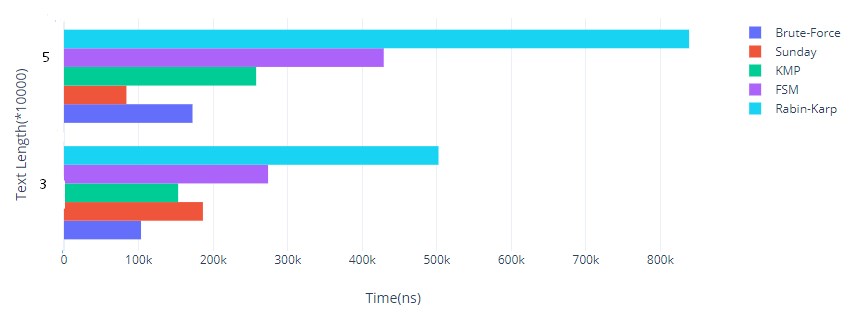
Graph



* The time complexity of the preprocessing phase in O(m) and constant space.
* The time complexity of the searching phase in O(mn).
* O(n+m) expected running time.

## 6-Conclusion





FSM is the slowest algorithm, according to first graph. Sunday and Brute-Force is executed the algorithm stable.

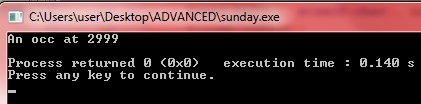
According to second graph, FSM is executed faster than Rabin-Karp because number of character variations is increased.

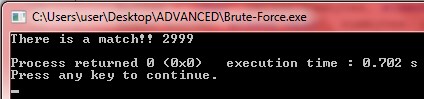
# WACKY RACES

## 1-Sunday vs Brute-Force

The Brute-Force algorithm doesn’t work well in cases where we see many matching characters followed by a mismatching character.

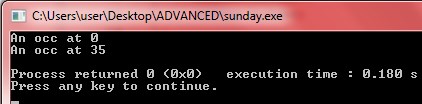
If text is chosen ‘aaaa........ab’ and pattern is chosen ‘ab’, The Sunday algorithm can search faster.

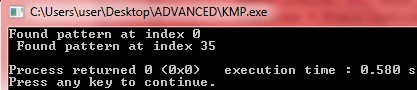




## 2-Sunday vs KMP

KMP have the worst case on where text is ‘((n-1)time)ab((n-1)time)ab….’,and pattern ‘(n times)a’ but still the time complexity of the KMP is O(m+n). That’s why Sunday algorithm can work faster.





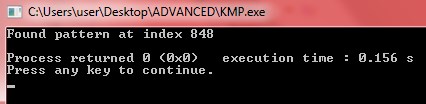
## 

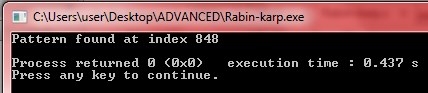
## 3-KMP vs Rabin-Karp

The Rabin-Karp algorithm calculates hash value for the pattern and text to compare if there is a match or not.

If pattern hash equals to text, the algorithm will look if the character match or not. There can be situation which is hash value equals but characters in the text not match. In this situation, The Rabin-Karp algorithm can work slowly.

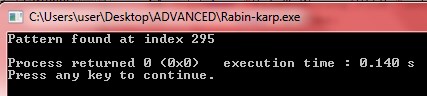
For instance, if text is chosen ‘CCACCAAECCACCAAE....DBA’, and pattern is chosen ‘DBA’. KMP algorithm can work fast.

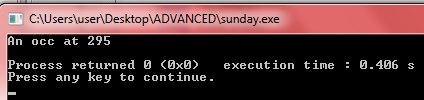




## 4-Rabin-Karp vs Sunday

If text is chosen ‘xxxx…zxxxxx...’ and pattern is chosen ‘xxxxxxxz’, Rabin-Karp can work fast.



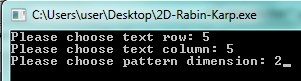


# JEWISH-STYLE CARP

Jewish style carp is calculates a hash value both pattern and text. If hash value of the pattern equals to text hash, it execute a brute-force to control character match. There can be situation which is hash value equals but character not match. That’s why it controls with brute-force algorithm.

In the code, it takes a row number and column number from user for create m\*n dimensional text array and then it ask to user for pattern dimension to create k\*k array.

Text index takes value randomly from randomlatter which is keeping alphabet.



The algorithm fill the pattern indices k\*k from top-right of the text. After that, it calculates a hash value for pattern and text to compare value match. İf there is a match it will run Brute-Force algorithm to check character match. When it finds a match, it will show the start indices.

