UNIVERSITY OF ŁÓDŹ



ADVANCED ALGORITHMS

PATTERN MATCHING ALGORITHMS

OGUZHAN OSMA

table of contents

[Introduction 4](#_Toc38618287)

[CHAPTER 1 – STRING MATCHING 5](#_Toc38618288)

[CHAPTER 2 - ALGORITHMS 6](#_Toc38618289)

[Brute Force 6](#_Toc38618290)

[Rabin-Karp Algorithm 8](#_Toc38618291)

[KMP Algorithm 9](#_Toc38618292)

[FSM Algorithm 13](#_Toc38618293)

[SUNDAY Algorithm 16](#_Toc38618294)

[CHAPTER 3 – COMP. OF ALGORTHMS(PART 1) 17](#_Toc38618295)

[CHAPTER 4 – WACKY RACES (PART 2) 23](#_Toc38618296)

[1-SUNDAY VS BRUTE FORCE 23](#_Toc38618297)

[2-SUNDAY VS KMP 24](#_Toc38618298)

[3-KMP vs RABIN-KARP 24](#_Toc38618299)

[4-RABIN-KARP vs SUNDAY 25](#_Toc38618300)

[CHAPTER 5 – JEWISH-STYLE CARP(PART 3) 26](#_Toc38618301)

[REFERENCES 27](#_Toc38618302)

lıst of fıgures

[Figure 1 6](#_Toc40190406)

[Figure 2 – Brute Force First Step 7](#_Toc40190407)

[Figure 3- Brute Force Second Step 7](#_Toc40190408)

[Figure 4 – Brute Force Third Step 7](#_Toc40190409)

[Figure 5 – Brute Force Fourth Step 8](#_Toc40190410)

[Figure 6 – Brute Force Fifth Step 8](#_Toc40190411)

[Figure 7 – Brute Force Last Step 8](#_Toc40190412)

[Figure 8- Rabin-Karp Preprocessing 9](#_Toc40190413)

[Figure 9 – Rabin-Karp 9](#_Toc40190414)

[Figure 10 – Rabin Karp 9](#_Toc40190415)

[Figure 11- Rabin Karp Matching Hashes 9](#_Toc40190416)

[Figure 12 – Rabin Karp Match 10](#_Toc40190417)

[Figure 13 – KMP Prefix Table 11](#_Toc40190418)

[Figure 14 – KMP Step 1 11](#_Toc40190419)

[Figure 15 - KMP Step 2 11](#_Toc40190420)

[Figure 16 - KMP Step 3 12](#_Toc40190421)

[Figure 17 - KMP Step 4 12](#_Toc40190422)

[Figure 18 - KMP Step 5 12](#_Toc40190423)

[Figure 19 - KMP Step 6 13](#_Toc40190424)

[Figure 20 - KMP Step 7 13](#_Toc40190425)

[Figure 21 - KMP Step 8 13](#_Toc40190426)

[Figure 22 - KMP Final Step 14](#_Toc40190427)

[Figure 23 – FSM Algorithm 14](#_Toc40190428)

[Figure 24 – FSM Algorithm 15](#_Toc40190429)

[Figure 25 – FSM Algorithm 15](#_Toc40190430)

[Figure 26 – FSM Algorithm 15](#_Toc40190431)

[Figure 27 - FSM Algorithm 16](#_Toc40190432)

[Figure 28 - FSM Algorithm 16](#_Toc40190433)

[Figure 29 - FSM Algorithm Diagram 16](#_Toc40190434)

[Figure 30 - FSM Algorithm Final 17](#_Toc40190435)

[Figure 31 – Sunday Algorithm 17](#_Toc40190436)

[Figure 32- Brute Force 100-1000 Chars 18](#_Toc40190437)

[Figure 33 – Brute Force 10K-100K Chars 18](#_Toc40190438)

[Figure 34 – Sunday 100-1000 Chars 19](#_Toc40190439)

[Figure 35 – Sunday 10K-100K Chars 19](#_Toc40190440)

[Figure 36 – KMP 100-1000 Chars 19](#_Toc40190441)

[Figure 37 – KMP 10K-100K Chars 20](#_Toc40190442)

[Figure 38 – FSM 100-1000 Chars 20](#_Toc40190443)

[Figure 39 – FSM 10K-100K Chars 20](#_Toc40190444)

[Figure 40 – Rabin-Karp 100-1000 Chars 21](#_Toc40190445)

[Figure 41 – Rabin-Karp 10K-100K Chars 21](#_Toc40190446)

[Figure 42 – 100-1000 Chars Comparison 22](#_Toc40190447)

[Figure 43 - 10K-100K Chars Comparison 22](#_Toc40190448)

[Figure 44 – QPC vs CLOCK\_MONOTONIC 23](#_Toc40190449)

[Figure 45 – Sunday vs Brute Force(Sunday) 24](#_Toc40190450)

[Figure 46 - Sunday vs Brute Force(Brute Force) 24](#_Toc40190451)

[Figure 47 - Sunday vs KMP(KMP) 24](#_Toc40190452)

[Figure 48 – Sunday vs KMP(Sunday) 25](#_Toc40190453)

[Figure 49 – KMP vs Rabin-Karp(KMP) 25](#_Toc40190454)

[Figure 50 - KMP vs Rabin-Karp(Rabin-Karp) 25](#_Toc40190455)

[Figure 51 – Rabin-Karp vs Sunday(Rabin-Karp) 26](#_Toc40190456)

[Figure 52- Rabin-Karp vs Sunday(Sunday) 26](#_Toc40190457)

[Figure 53 – Jewish Style Carp 26](#_Toc40190458)

# ıntroductıon

String matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. Why do we need string matching? String matching is used in almost all the software applications straddling from simple text editors to the complex NIDS.[1]

# chapter 1 – STRING MATCHING

Text-editing programs frequently need to ﬁnd all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efﬁcient algorithms for this problem—called “string matching”—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to ﬁnd Web pages relevant to queries.[2]

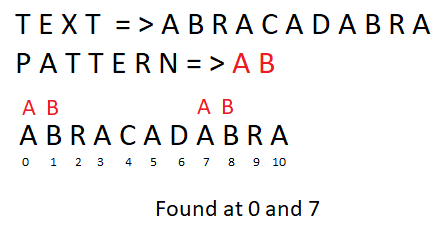


Figure 1

# CHAPTER 2 - ALGORITHMS

I’ll go with just one example through to explain algorithms. I assume that we’ve text which is “AAAB” and the pattern which we want to search is “AB”.

## bRute force

The brute force algorithm firstly searchs the first char of pattern in the text. If the first char equals to the char of text then it searchs the second char after the first equation if the pattern has more than 1 char.

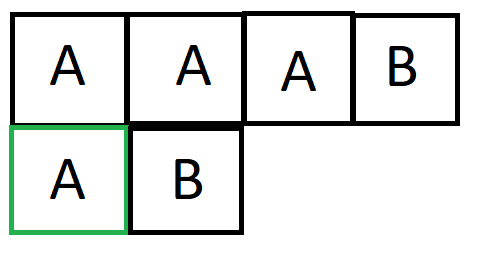


Figure 2 – Brute Force First Step

First char of pattern has been checked and it’s same with char of text. Next step is checking second char of pattern is same with text’s char or not.

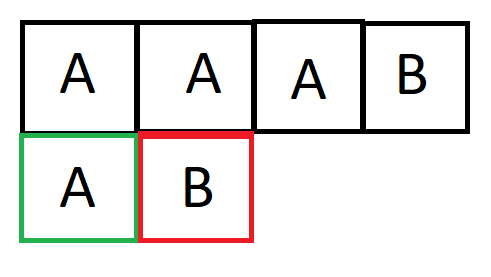


Figure 3- Brute Force Second Step

We checked and it didn’t match. Now we have to shift pattern and repeat first step.

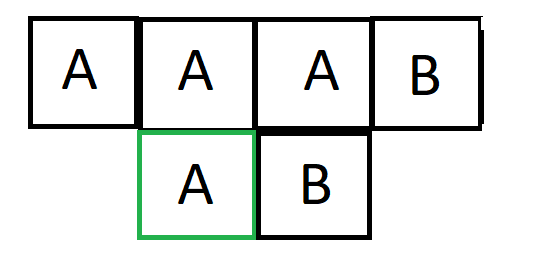


Figure 4 – Brute Force Third Step

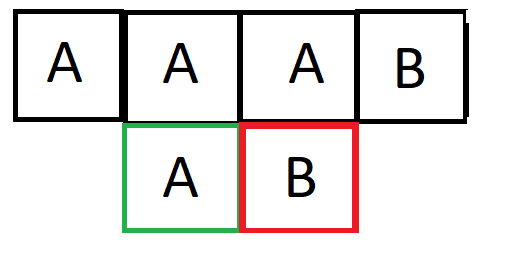


Figure 5 – Brute Force Fourth Step

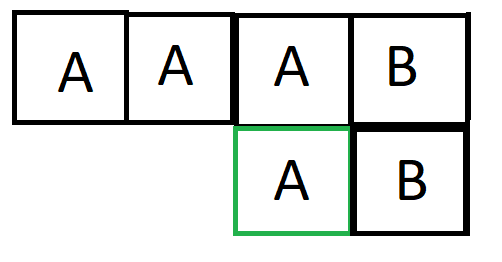


Figure 6 – Brute Force Fifth Step

As a last step we checked first char again and we’ve to check for second one.

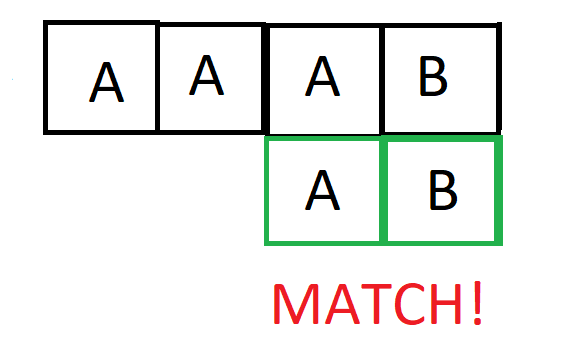


Figure 7 – Brute Force Last Step

After checking last char we saw that it’s matching. This is how brute force works.

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

|  |
| --- |
| txt[] = "AAAAAAAAAAB";  pat[] = "FA"; |

* When all characters of the text and pattern are same worst case occurs.

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

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

|  |
| --- |
| txt[] = "AAAAAAAAAAAAAAAAAB";  pat[] = "AAAAB"; |

The number of comparisons in the worst case is O(m\*(n-m+1)).

## Rabin-karp algorıthm

The main logic of this algorithm is comparing hash values of datas. If there is a match in hashes then it starts to compare characters. Let’s see with example.

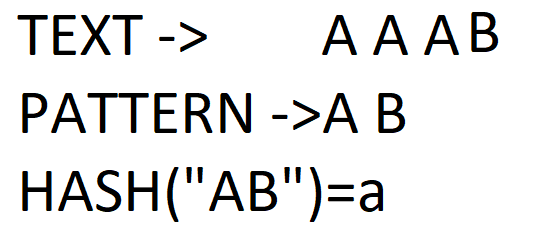


Figure 8- Rabin-Karp Preprocessing

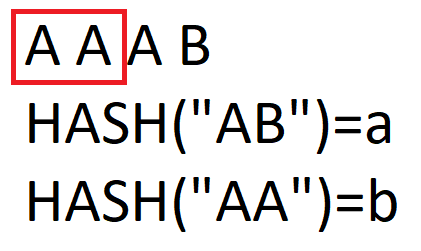


Figure 9 – Rabin-Karp

As you can see above the algorithm coumputed hash value of the string which has same length with pattern and it didn’t match. The next step is shifting.

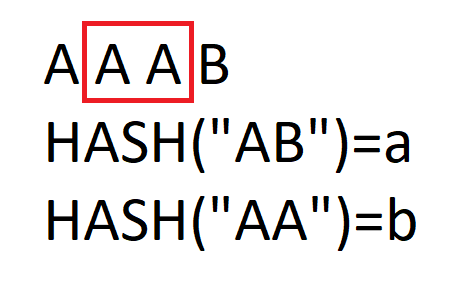


Figure 10 – Rabin Karp

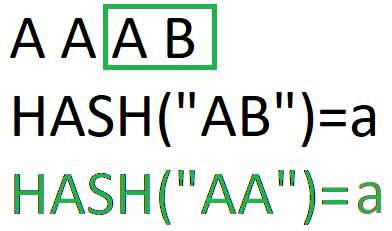


Figure 11- Rabin Karp Matching Hashes

Above you can see the algorithm have found the matching hash values. After that it’ll check the characters are same or not.

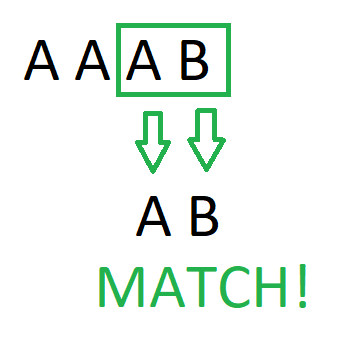


Figure 12 – Rabin Karp Match

When the whole chars are same with pattern, pattern is being found.

Preprocessing O(m) time complexity with constant space. The preprocessing phase of the Karp-Rabin algorithm consists in computing hash(q). Searching in O(nm) time. If an equality is found, it is still necessary to check character by character. Expected run time is O(n+m)that is by number of text character comparisons.

## KMP ALGORITHM

**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].

**Searching Algorithm:**

* We start comparison of pat[j] with j = 0 with characters of current window of text.
* We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
* When we see a **mismatch**
  + We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
  + We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.
  + From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider above example to understand this. [3]

Let’s see with an example.

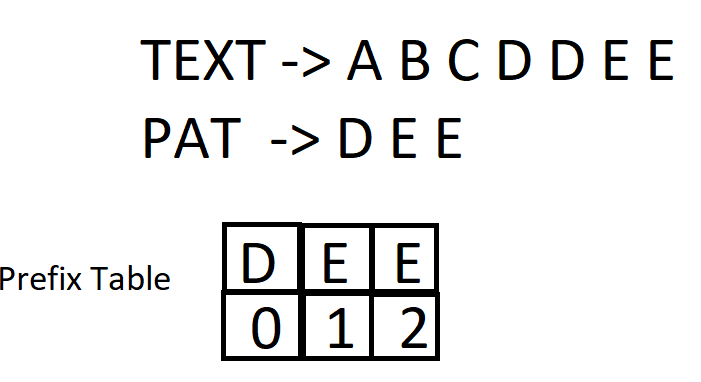


Figure 13 – KMP Prefix Table

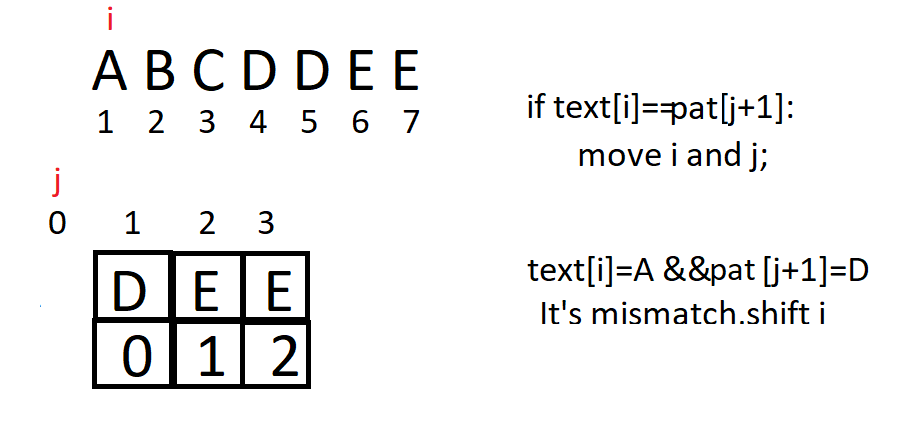


Figure 14 – KMP Step 1

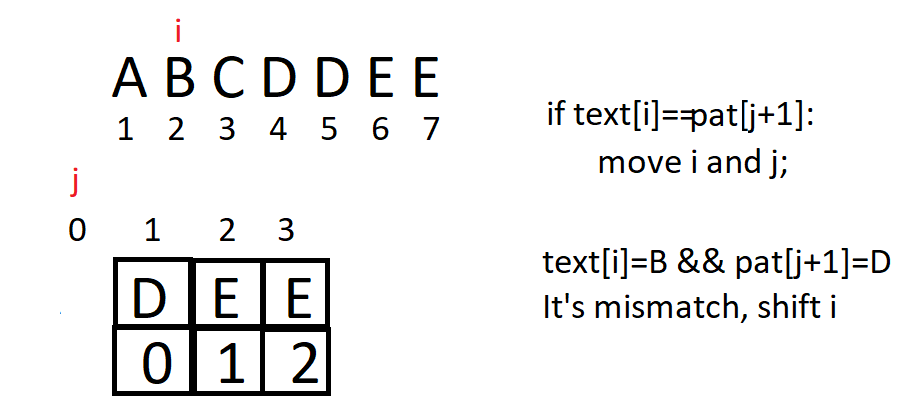


Figure 15 - KMP Step 2

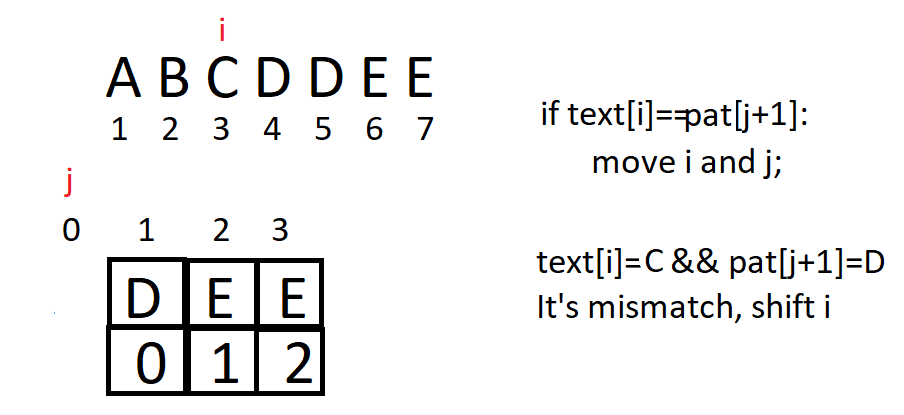


Figure 16 - KMP Step 3

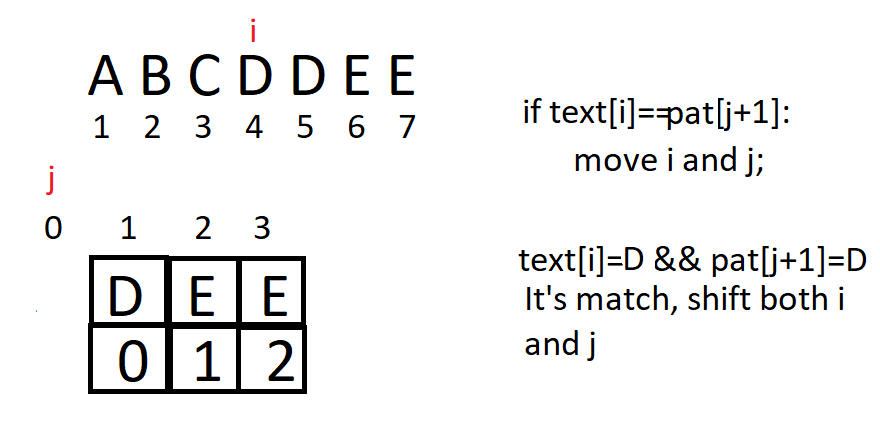


Figure 17 - KMP Step 4

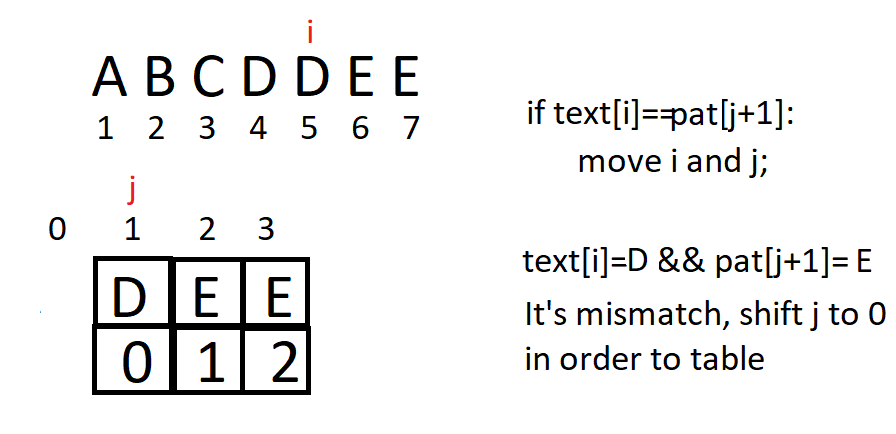


Figure 18 - KMP Step 5

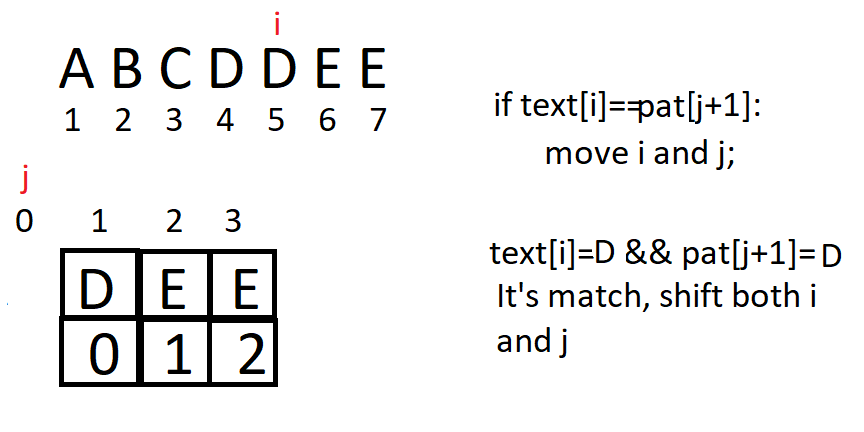


Figure 19 - KMP Step 6

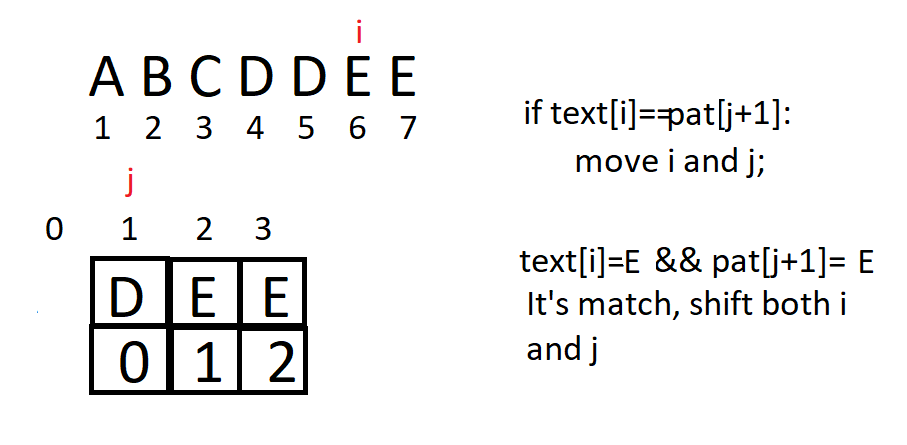


Figure 20 - KMP Step 7

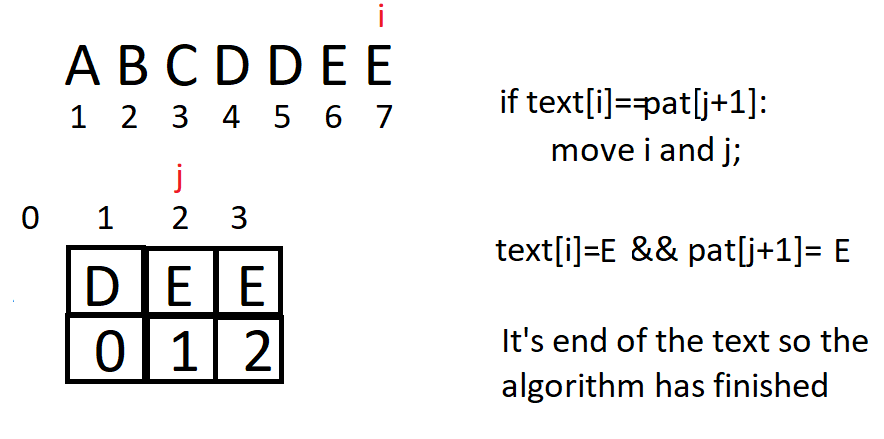


Figure 21 - KMP Step 8

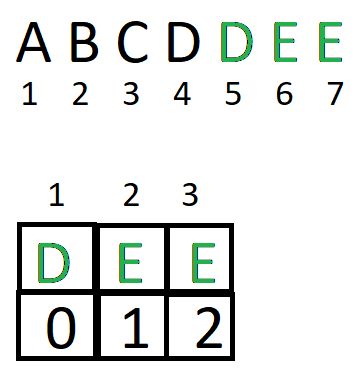


Figure 22 - KMP Final Step

## fsm algorıthm

In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to a new state. If we reach the final state, then the pattern is found in the text. The time complexity of the search process is O(n).

The time complexity of the computeTF() is O(m^3\*NO\_OF\_CHARS) where m is length of the pattern and NO\_OF\_CHARS is size of alphabet (total number of possible characters in pattern and text).

Let’s see the example.

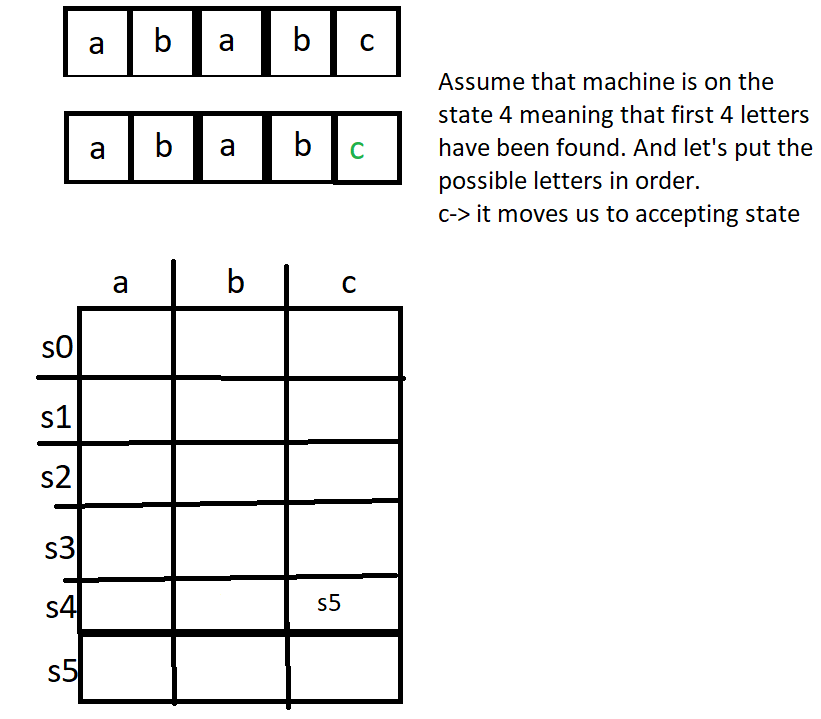


Figure 23 – FSM Algorithm

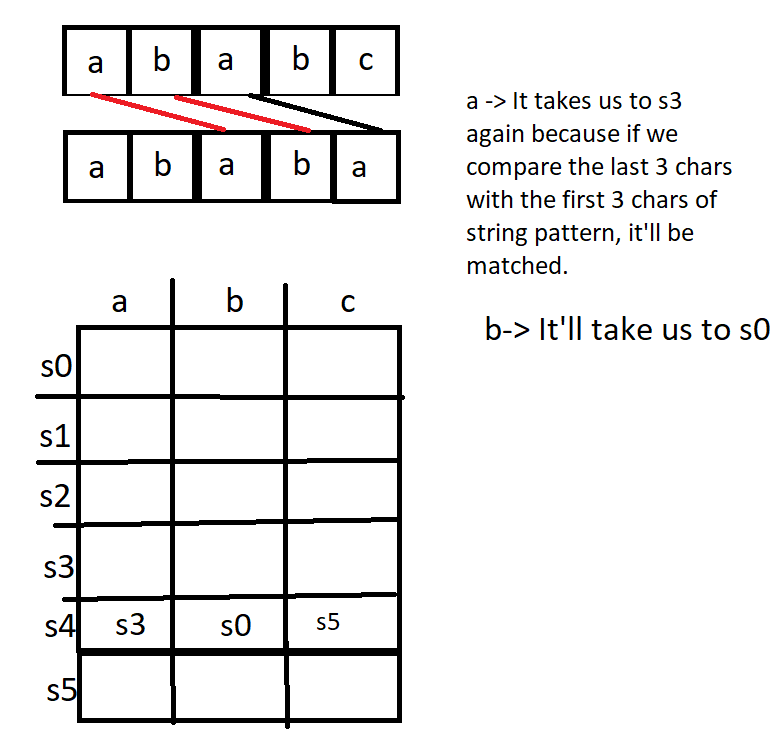


Figure 24 – FSM Algorithm

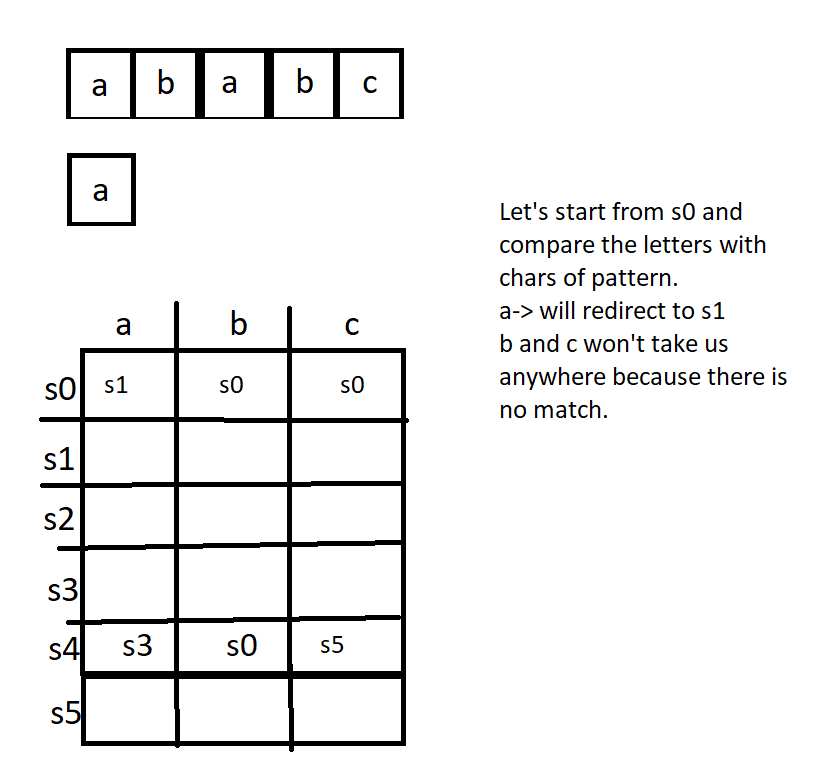


Figure 25 – FSM Algorithm

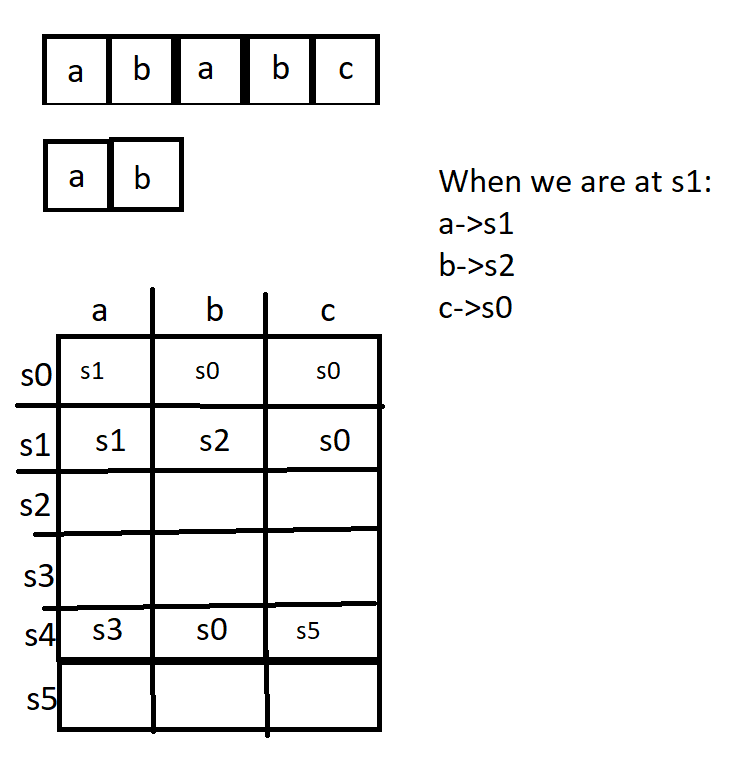


Figure 26 – FSM Algorithm

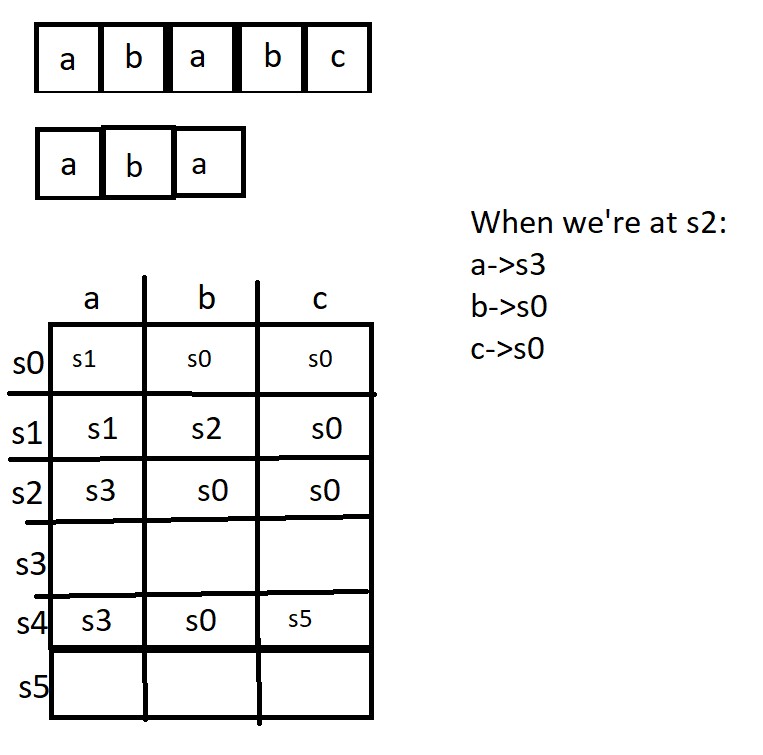


Figure 27 - FSM Algorithm

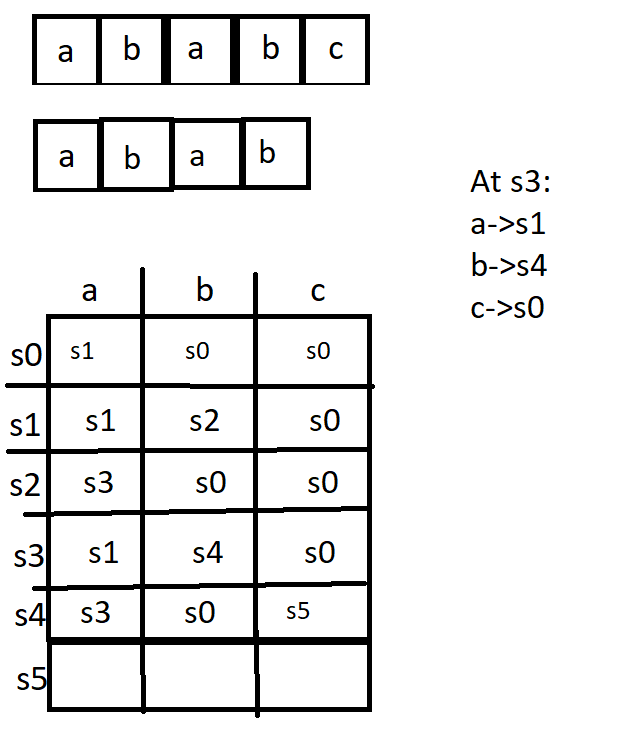


Figure 28 - FSM Algorithm

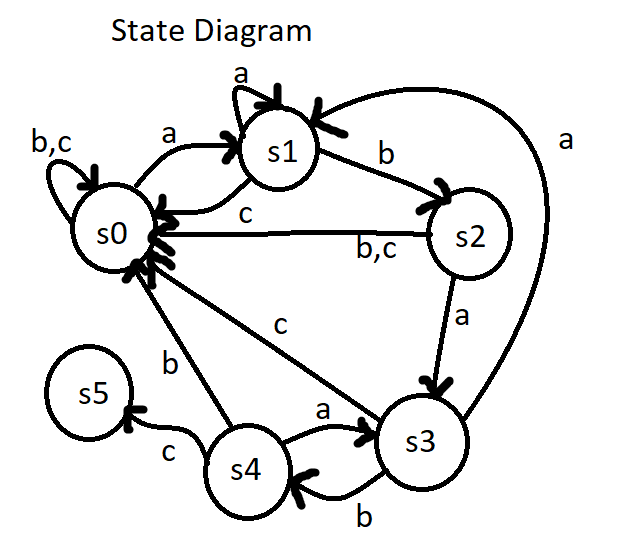


Figure 29 - FSM Algorithm Diagram

According to the table we can draw a diagram as above.

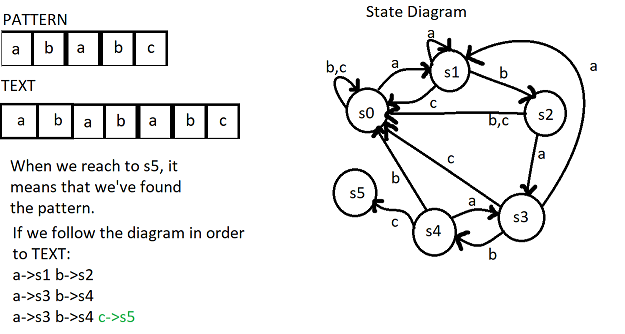


Figure 30 - FSM Algorithm Final

## SUNDAY ALGORITHM

Like the Boyer-Moore and the Horspool algorithm, the Sunday algorithm assumes its best case if every time in the first comparison a text symbol is found that does not occur at all in the pattern. Then the algorithm performs just O(n/m) comparisons.

In contrast to the Boyer-Moore and the Horspool algorithm the pattern symbols need not be compared from right to left. They can be compared in an arbitrary order. For instance, this order can depend on the symbol probabilities, provided they are known. Then the least probable symbol in the pattern is compared first, hoping that it does not match, so that the pattern can be shifted.[4]

Example:

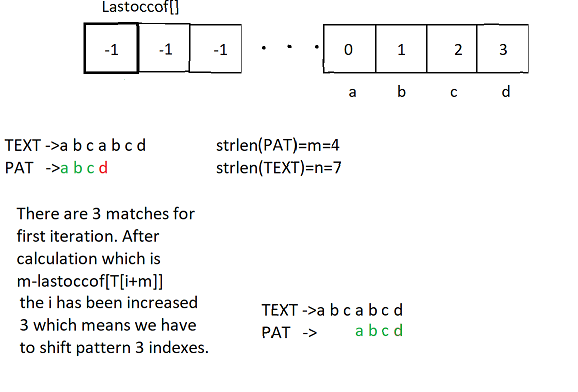


Figure 31 – Sunday Algorithm

# CHAPTER 3 - COMPARISON OF ALGORTHMS(PART 1)

**NOTE : In the all tests pattern was constant. The only thing which changes is text length.**

**BRUTE FORCE**

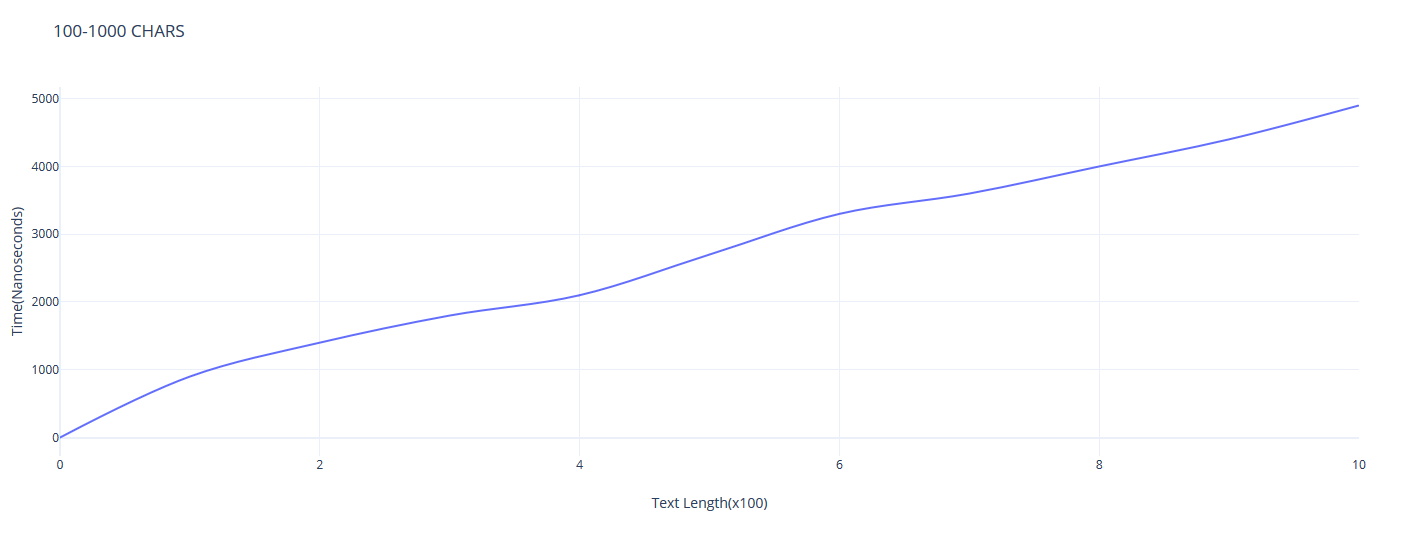


Figure 32- Brute Force 100-1000 Chars

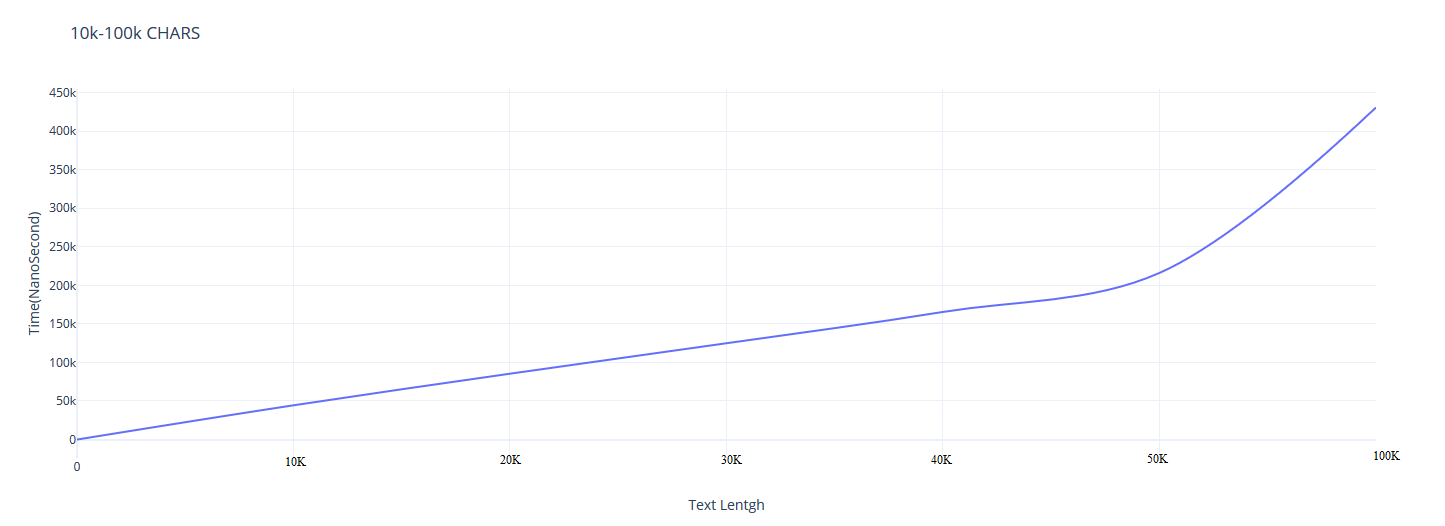


Figure 33 – Brute Force 10K-100K Chars

Brute Force algorithm gave a good result in 100-1000 chars because when the text length is short and pattern is short too, brute force is getting effective.

**SUNDAY**

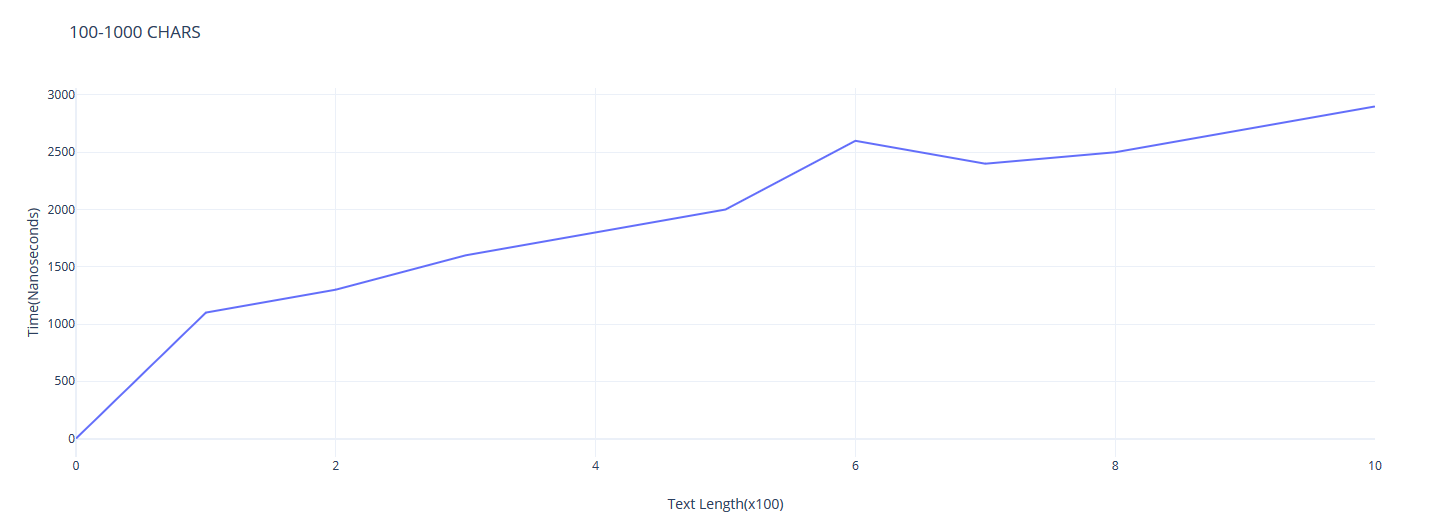
****

Figure 34 – Sunday 100-1000 Chars

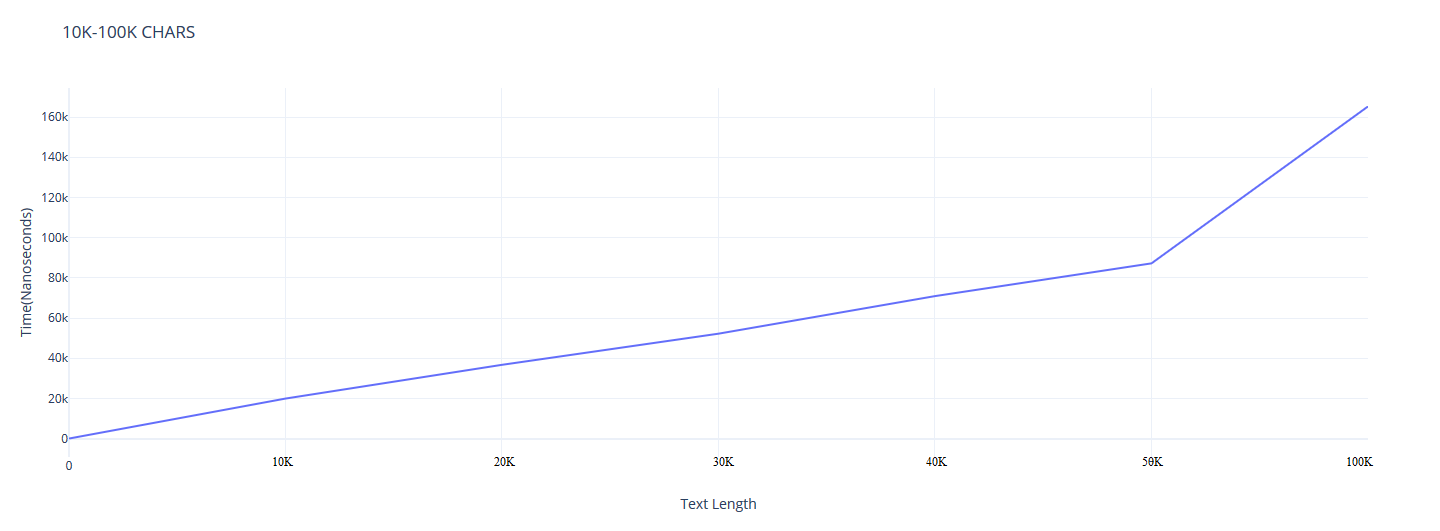


Figure 35 – Sunday 10K-100K Chars

**KMP**

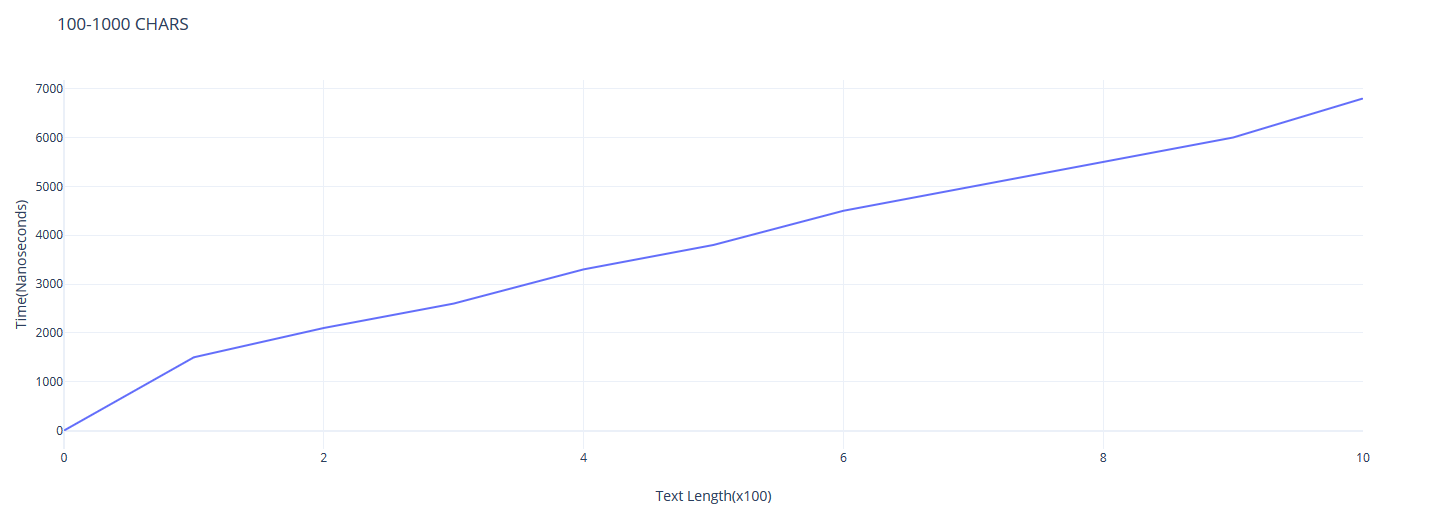
****

Figure 36 – KMP 100-1000 Chars

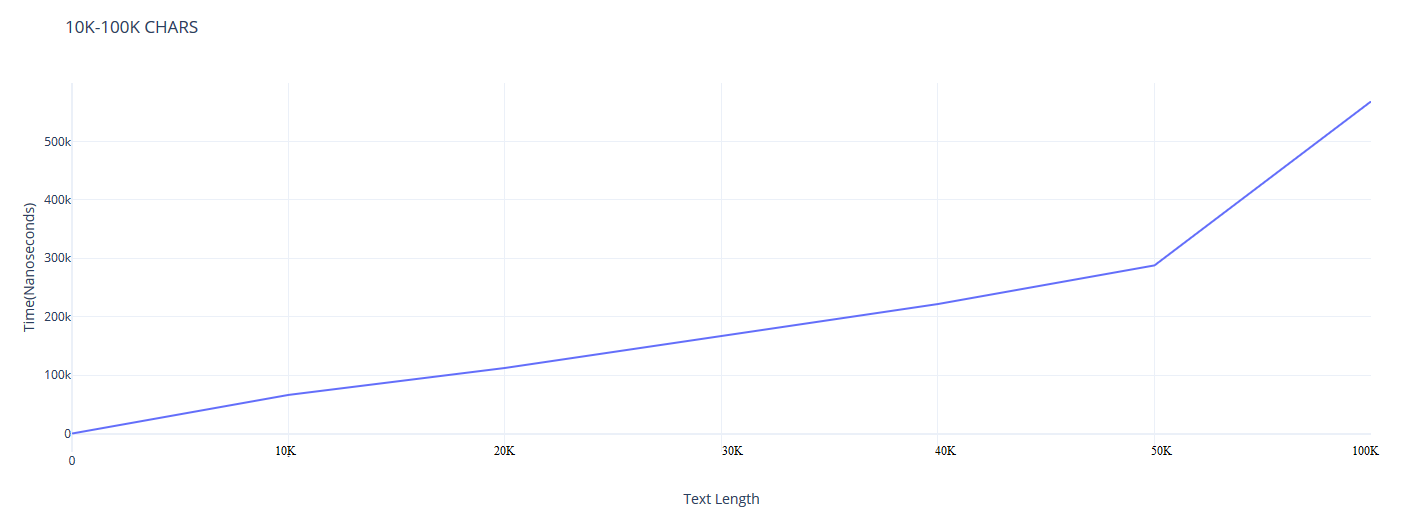


Figure 37 – KMP 10K-100K Chars

**FSM**

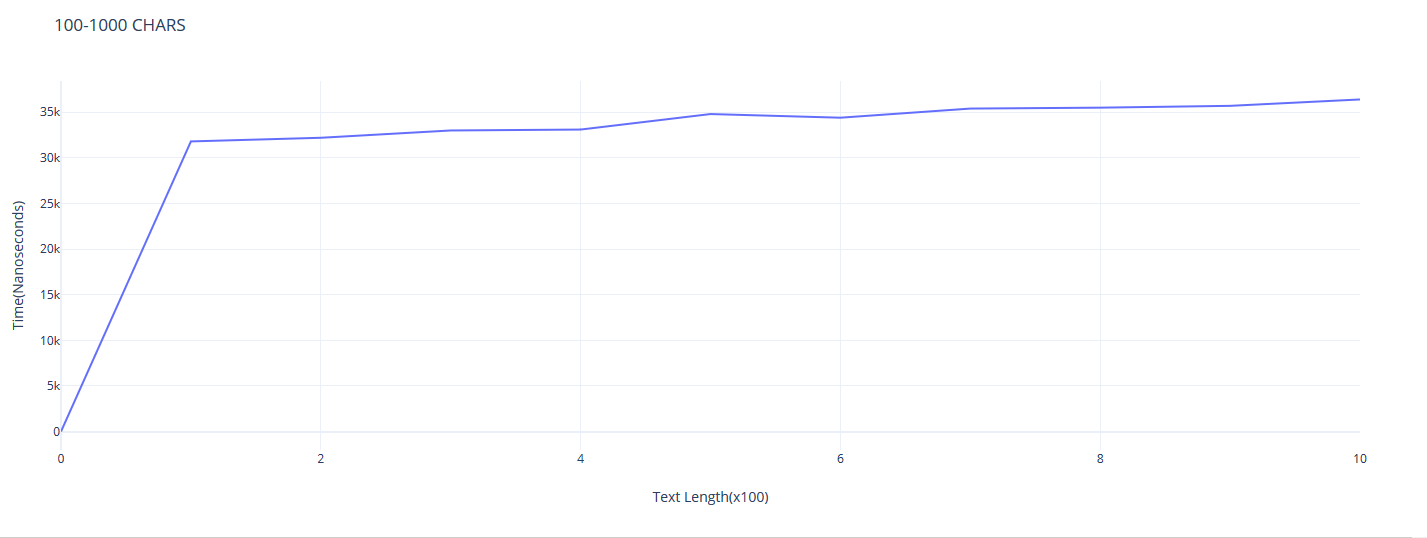
****

Figure 38 – FSM 100-1000 Chars

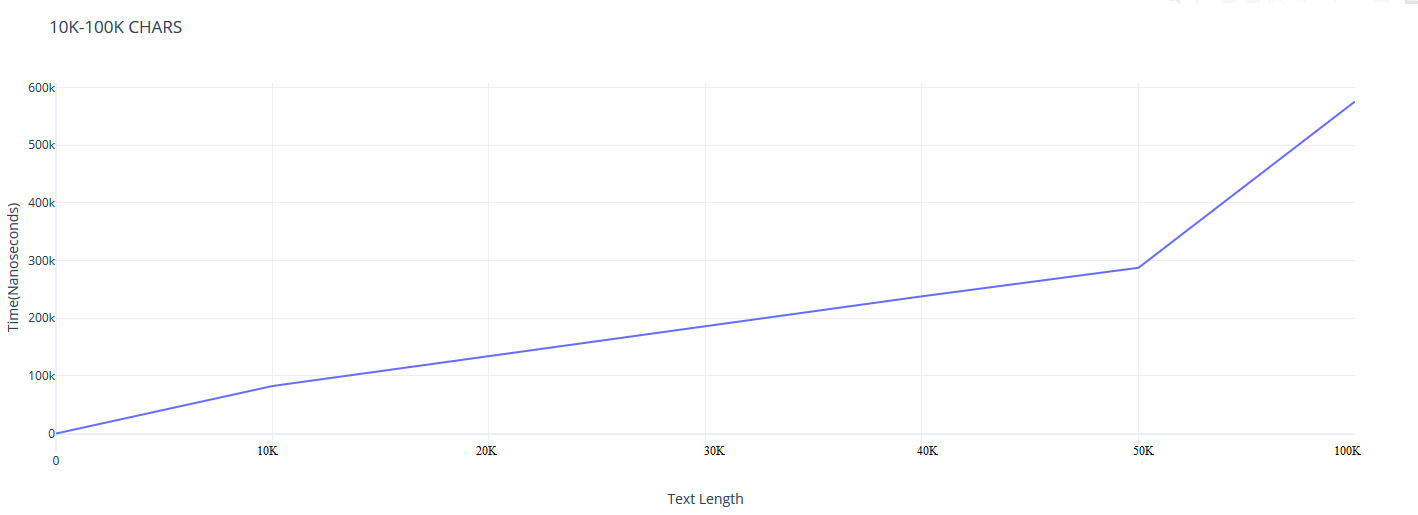


Figure 39 – FSM 10K-100K Chars

**RABIN-KARP**

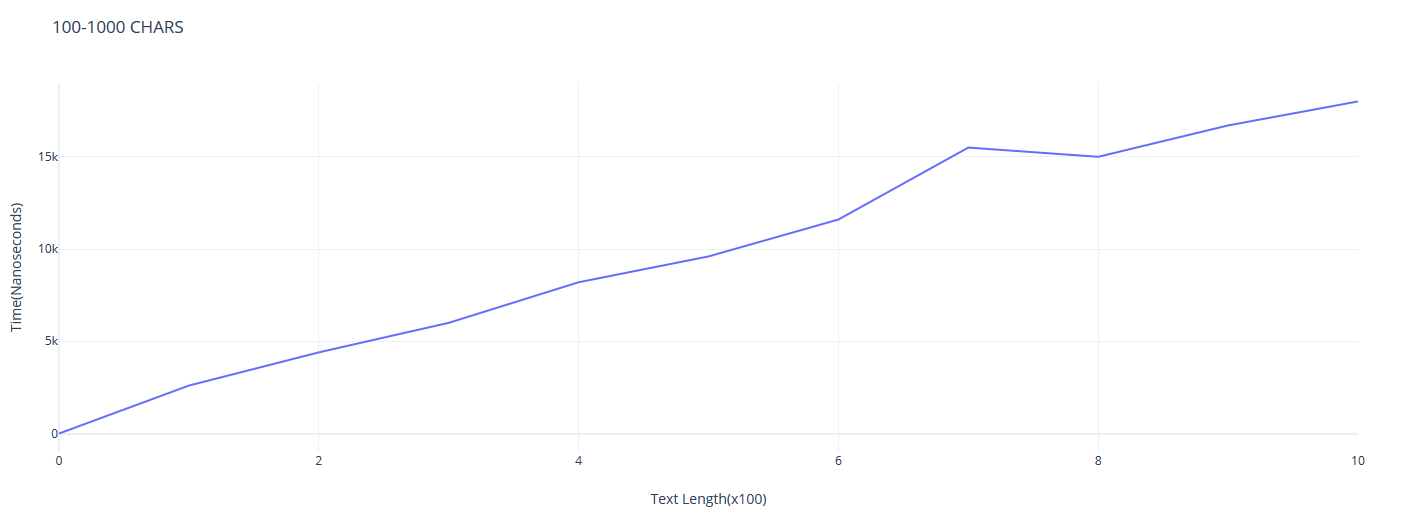
****

Figure 40 – Rabin-Karp 100-1000 Chars

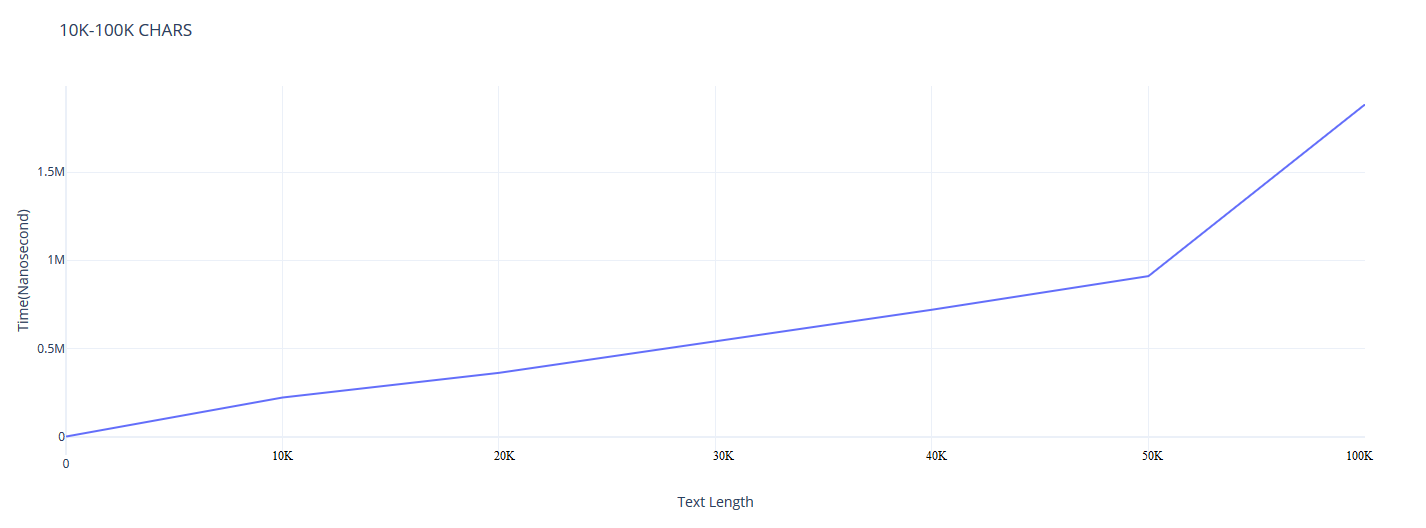


Figure 41 – Rabin-Karp 10K-100K Chars

All algorithms except FSM had more efficiency in 100-1000 chars comparison. When the text gets longer any algorithm should be chosen except FSM for that test. If we look the average, we can say that Sunday was the fastest one. The special feature of Sunday Algorithm is on preprocessing. Creation of array which, for each symbol of the alphabet, stores the position of its rightmost occurrence in the pattern. In 10K-100K comparison Rabin-Karp was the slowest one. It probably caused because of hash calculation. RK made a lot of calculations in those tests. KMP was stable one.

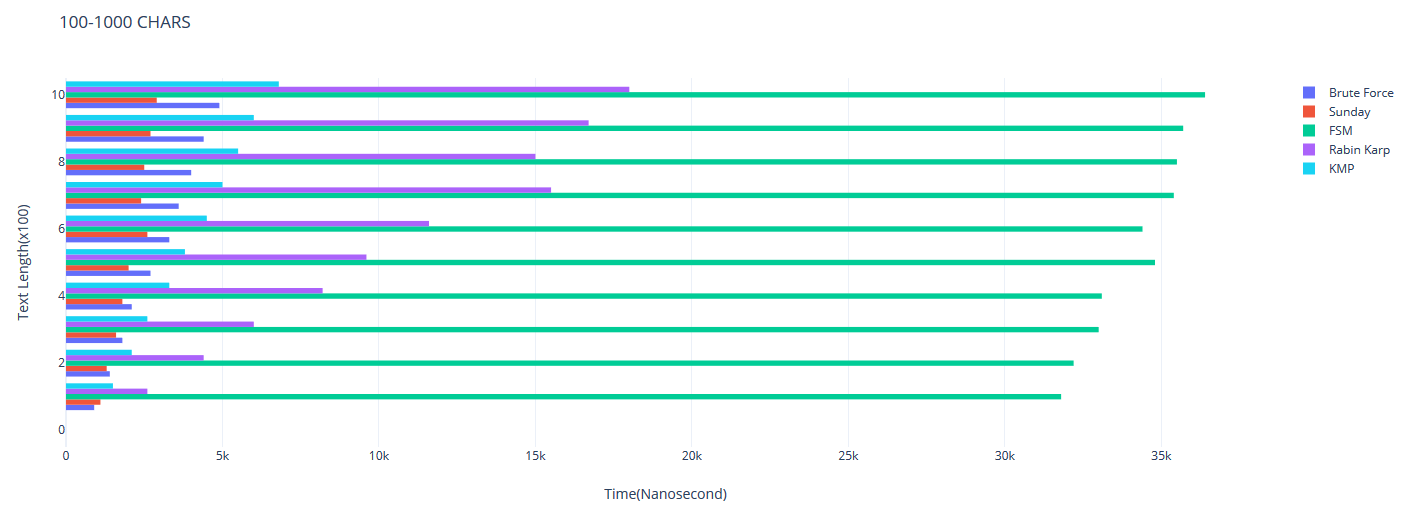


Figure 42 – 100-1000 Chars Comparison

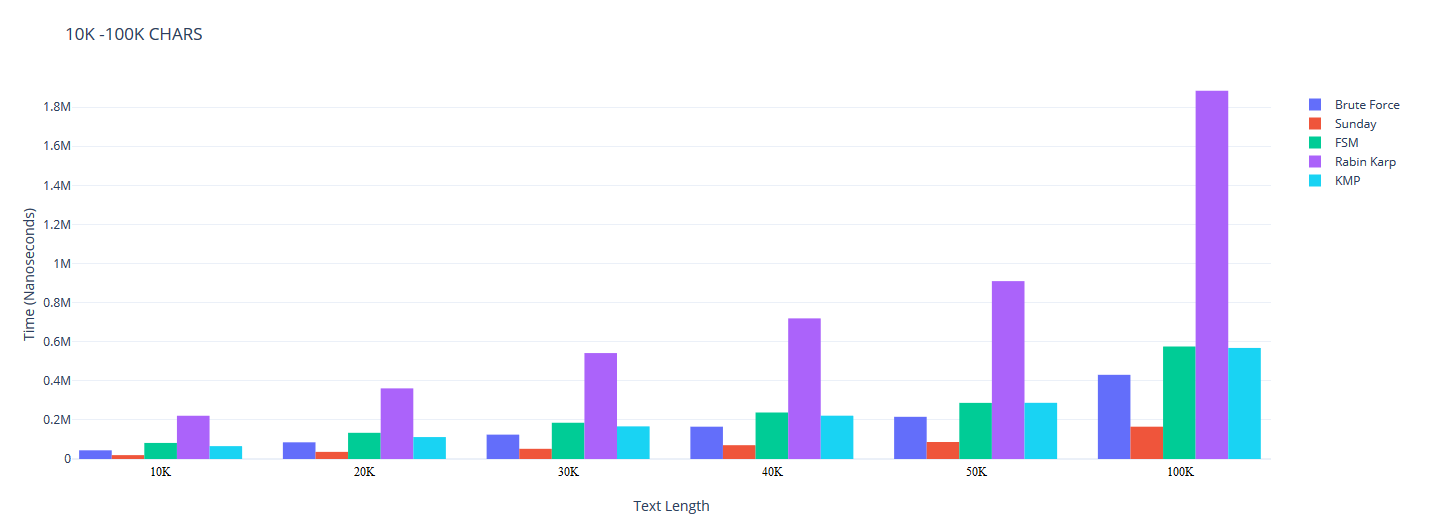


Figure 43 - 10K-100K Chars Comparison

FSM was generally the slowest one but when the text length gets longer, Rabin Karp became slower. FSM was faster in 10K-100K comparison because as you can see in the code NO\_OF\_CHARS is 256. When the text length gets longer, in FSM accessible number of chars are getting increased. Generally KMP and Sunday kept their stability in every tests. The result is not constant if we would change the pattern length, it’d give us different results. We can say that if the text is short, we can use Sunday and Brute Force. If the text is quite long and if it consist many chars so FSM and KMP would be good choice too.

**Comparison of QueryPerformanceCounter and CLOCK\_MONOTONIC**

For Brute Force I made comparison of time measurments. Their precisions were close.

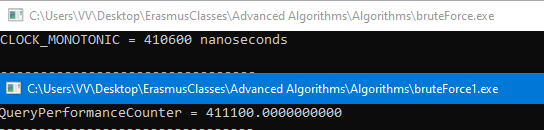


Figure 44 – QPC vs CLOCK\_MONOTONIC

Both calculations have been made in nanosecond and every circumstances were equal.

# CHAPTER 4 – WACKY RACES (PART 2)

### 1-sUNDAY VS BRUTE FORCE

Specific Text = aaaa…aaaaab

Specific Pattern =aa…aab

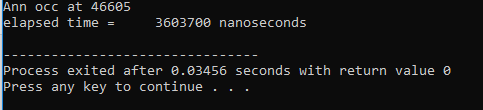


Figure 45 – Sunday vs Brute Force(Sunday)

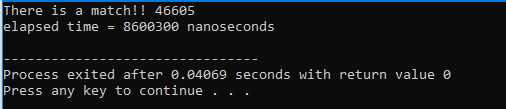


Figure 46 - Sunday vs Brute Force(Brute Force)

### 2-sUNDAY VS KMP

Specific Text = xxxx…xxyxxx…xxxy…

Specific Pattern = xxx…xxx

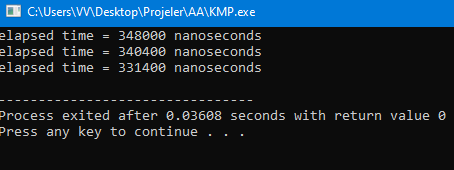


Figure 47 - Sunday vs KMP(KMP)

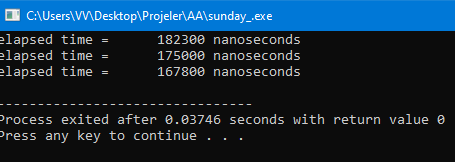


Figure 48 – Sunday vs KMP(Sunday)

### 3-KMP vs rabın-karp

Specific Text = AAA…AAA(500)

Specific Pattern = AAA…AAA(450)

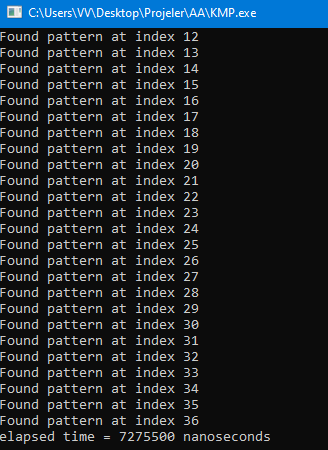


Figure 49 – KMP vs Rabin-Karp(KMP)



Figure 50 - KMP vs Rabin-Karp(Rabin-Karp)

### 4-rabın-karp vs sunday

Specific Text = aaa…aaaaacaaaa…

Specific Pattern = aaa…aaac

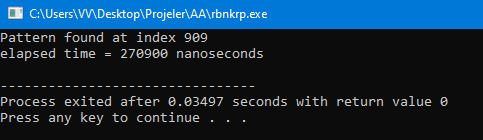


Figure 51 – Rabin-Karp vs Sunday(Rabin-Karp)

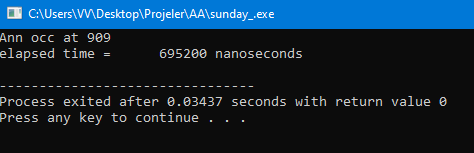


Figure 52- Rabin-Karp vs Sunday(Sunday)

# CHAPTER 5 – jewısh-style carp(part 3)

Rabin-Karp algorithm uses a hash function to speed up the search. Instead of using a string, we will use a **two-dimensional** array. The program only shows first index of found pattern. Instead of modulo operation, bitwise & mask has been used.

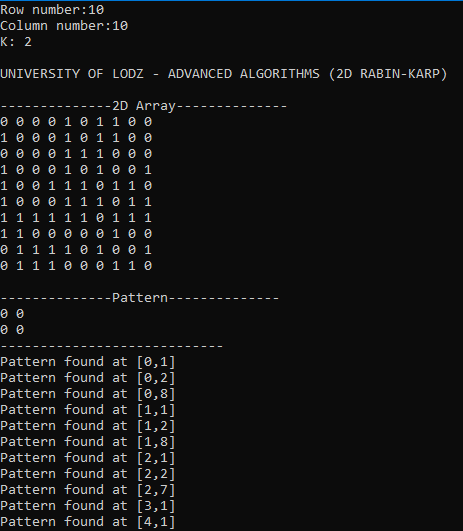


Figure 53 – Jewish Style Carp

# references

[1]: <https://www.slideshare.net/alokeparnachoudhury/string-matching-algorithm>

[2]: (CLRS) Thomas Cormen, et. al. "Intruduction to Alglortihms"

[3]: <https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

[4]: <https://www.inf.fh-flensburg.de/lang/algorithmen/pattern/sundayen.htm>