

Authors: **Cosmin-Gabriel IVAN 10LF322**

**Mihai MALOȘ 10LF322**

**Livia-Daniela TOMA 10LF323**

Scientific coordinator: **Lect. Dr. Alexandra BĂICOIANU**

Brașov, 2022 – 2023

**String matching**

**Knuth-Morris-Pratt algorithm**

-Data protection-

Table of contents

[BACKGROUND 3](#_Toc126099981)

[MAIN FEATURES 4](#_Toc126099982)

[DESCRIPTION OF THE ALGORITHM 5](#_Toc126099983)

[How does the algorithm work? 5](#_Toc126099984)

[Advantages and disadvantages of the KMP algorithm 8](#_Toc126099985)

[Applications of the KMP Algorithm 8](#_Toc126099986)

[KMP algorithm compared to the Naive string matching 8](#_Toc126099987)

[Similarities and differences between KMP and Naive algorithms 10](#_Toc126099988)

[Efficiency of the Knut-Morris-Pratt algorithm – multiple tests 12](#_Toc126099989)

[Conclusion 13](#_Toc126099990)

[BIBLIOGRAPHY 14](#_Toc126099991)

BACKGROUND

Donald Knuth, born in 1938, is one of the American’s computing pioneers, a mathematician and professor at Stanford University, known for his influential multi-volume work ”The Art of Programming” and the invention of TeX, the text-formatting language[[1]](#footnote-1).

The reason why the famous algorithm is called ”Knuth-Morris-Pratt” is because each of the three programmers found the algorithm that worked in linear time, however they didn’t discover it at the same time. Jim Morris was the first one to notice that there is a cleverer way of performing a search of a string in a larger string. He managed to develop the algorithm and due to the fact that the people he was working with did not understand the procedure, they removed it, so this brilliant idea became lost. A year or two later, Vaughan Pratt had also independently discovered the process, while studying a technical abstract problem about concatenation of palindromes. An aspect that influenced the path to success was Cook’s theorem about stack automaton, the demonstration that if you can solve a specific type of problem on a computer that is limited in its capability, then there has to be a faster way to do it on a regular computer. Donald Knuth used Cook’s theorem in order to solve a problem, and therefore, he managed to find a fast way to resolve the issues he was facing. From that specific point on, Knuth was convinced that automata theory can be useful for day-to-day programming. When the professor shared the information with Vaughan Pratt, while on a regular trip at Berkeley, they started to put all the thing together, and eventually came up with an efficient algorithm about string-searching. ‘’For the most part, the procedure KMP – Matcher follows from Finite – Automaton -Matcher”[[2]](#footnote-2) All these events led to all three programmers to define a new, particular, certain algorithm that could solve plenty of daily problems.



*Image 1. Interview with Donald Knuth – The Knuth-Morris-Pratt algorithm*

MAIN FEATURES

”In [computer science](https://en.wikipedia.org/wiki/Computer_science), string-searching algorithms, sometimes called string-matching algorithms, are an important class of [string algorithms](https://en.wikipedia.org/wiki/String_algorithms) that try to find a place where one or several [strings](https://en.wikipedia.org/wiki/String_(computer_science)) (also called patterns) are found within a larger string or text.” [[3]](#footnote-3)

String matching is a method that provides a feasible solution to a string-searching problem. Despite the first instinct, the needless scan of the entire text, the more practical way of performing the search is using a string-matching algorithm. This significant class of string algorithms attempt to find the position where one, or a sequence of strings, also called patterns, are found within a larger string or text.

Knuth, Morris and Pratt designed an exact string-matching algorithm, based on character comparison, whose main purpose is to find a specific pattern in a text. The comparisons are performed character bycharacter, from left to right. As for space and time complexity, the KMP algorithm runs the pre-processing phase in *O (m)* complexity and, regardless the system of signs, letter and characters, or its size, the searching phase has *O (n + m)* time complexity. [[4]](#footnote-4)

DESCRIPTION OF THE ALGORITHM

**“An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems.”[[5]](#footnote-5)**

How does the algorithm work?

**The pre-processing part of the KMP algorithm**: Also known as LPS, consists of a vector which finds longest proper prefix which is also suffix in the pattern for all strings who start with index 0.

Through the parameter ”pattern” which is a string and has been received from the header of the function which creates the table, there will be assigned a word of length m, of int type. A vector of int type elements, named LPS, is generated and it receives enough memory for the length of the word, m. Using the position of each character in the given word, excepting the first element, more specifically LPS[0], which is initialized from the beginning with 0, there are comparisons performed between pattern[j] with pattern[i]. To determine the stored values from the LPS vector, the next criteria are required:

LPS[i] = j + 1;

* pattern[j] = pattern[i] → i = i + 1;

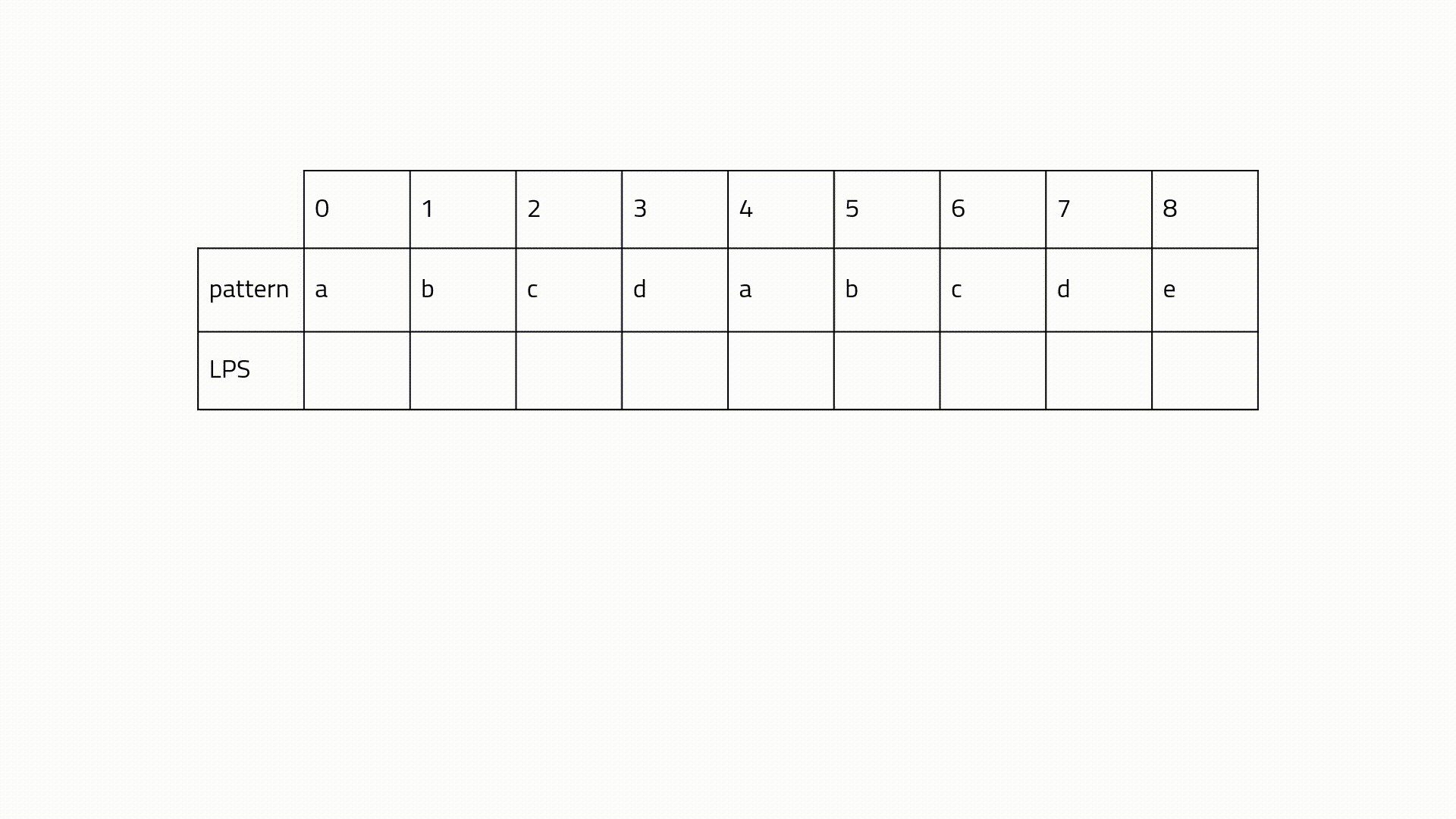
j = j + 1;

* pattern[j] ≠ pattern[i] && j = 0 → LPS[i] = 0;

i = i + 1;

j ≠ 0 → j = LPS[j-1];

All these requirements repeat until all characters from the parameter „pattern” have been covered. The value used for every character, more precise LPS[j], represents the length of the longest specific prefix which is also a suffix for the first ‘j’ characters, where 0 < j < n-1. Pre-processing includes the construction of the prefix table, but the catch is that it has to be performed in linear time *O(n)*, where n is the length of the pattern that generates the LPS vector.

In the following example, it is presented the construction of a LPS table, for the word „abcdabcde” of length n = 9.

The LPS table was created due to the partial matches that caused additional calculation time, therefore, using this prefix table, the extra work can be exempted, thus avoiding unnecessary comparisons. Through this idea the purpose is defined this KMP algorithm.

**The KMP matcher:** It receives two parameters of string type, that store the pattern which has to be respected in order to create the prefix table, of length m, and furthermore the text, of length n, in which the search for the elements from the first parameter is carried out. Similar to the naive algorithm, both of them operate the same way, but the difference is the avoidance of the partial matches. To gain efficiency, KMP uses pattern pre-processing, so that it manages to decide what to overlook when discrepancies occur. In order to look through the entire text, and simultaneously search for the characters from the pattern, the following criteria are used:

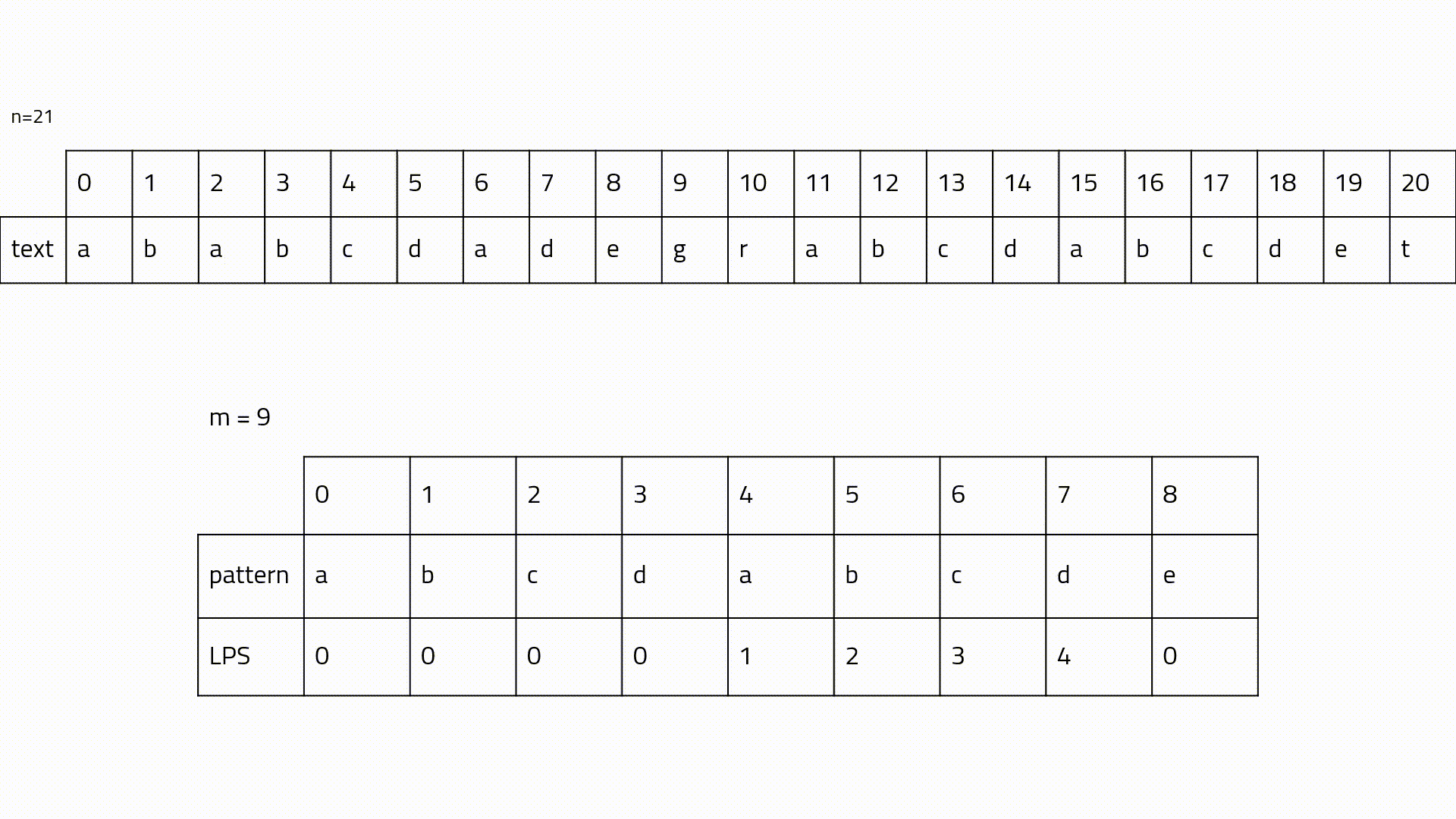
* pattern[j] = text[i] → i = i + 1;

j = j + 1;

* j = m → show i – m;
* ( i < n ) && ( pattern[j] ≠ text[i] ) && ( j = 0 ) → i = i + 1;

( j ≠ 0 ) → j = LPS[j – 1];

All these features repeat until all characters from parameter „text” have been covered.

****In the following example, the search for the previous word “abcdabcde” is performed in the text „ababcdadegrabcdabcdet”, which has the length n = 21, and at the end, it is displayed the position of the first character where it was found given word.

Advantages and disadvantages of the KMP algorithm

A very obvious advantage of the KMP algorithm is it's time complexity. It's very fast as compared to any other exact string matching algorithm. The KMP has the nice advantage that it is guaranteed worst-case efficient. No worse case or accidental inputs exist here. Moreover, if the string has to be found in a larger input, the search is easier and more efficiently performed due to the linear time complexity .

On the other hand, the algorithm doesn’t work well when the size of the alphabets increases. Due to this issue, more and more error occurs.[[6]](#footnote-6) Another disadvantage of the KMP algorithm is that it is very complex to understand.

Applications of the KMP Algorithm

* Checking for Plagiarism in documents etc
* Bioinformatics and DNA sequencing
* Digital Forensics
* Spelling checkers
* Spam filters
* Search engines, or for searching content in large databases
* Intrusion detection system

KMP algorithm compared to the Naive string matching

**Naive string matching**

One approach for searching a specific pattern in the input string can be performed using two loops. This perspective tests every possible placement of the pattern string in the input text, in order to determine whether the current positioning matches the input pattern or not. In case the condition is fulfilled, then the program should return the first index of the matching window of input text. Otherwise, if all the possible positioning does not match with the input string, then the program would return -1

As for this method, all the possible placements of input pattern[1 ... m] are compared to the input text [1 ... n].

In the naive string searching, the entire text is scanned until a match with the input pattern is found. . If the first character matches the input string text, we are searching the window of size m.

However, we are performing unnecessary computations, as in we’re repeatedly looking for a match in elements that we have already traversed, and this makes the algorithm slow. !!! site . The time complexity turns out to be *O(n-m+1)* and the space complexity of the algorithm comes out to be *O(1)*.

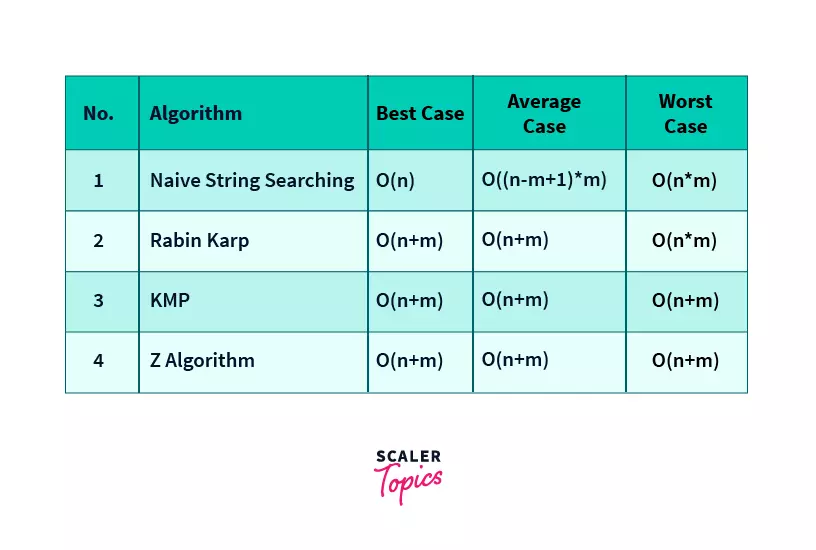


Figure: Time Complexities of String Searching Algorithms

<https://www.scaler.com/topics/data-structures/z-algorithm/>

Similarities and differences between KMP and Naive algorithms

Similarities:

* In order to accomplish the comparisons, there are two string pointers used.
* The two indexes which compare character by character start from left to right.
* Both algorithms are used to identify a character sequence from a multitude of words.
* The returned value represents the starting index of the match, or the indication that no match was found[[7]](#footnote-7).

Differences:

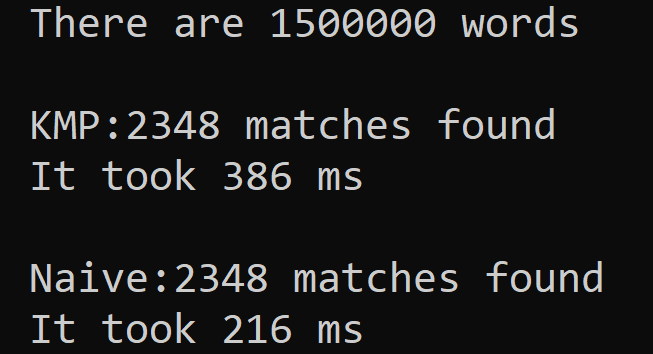
|  |  |
| --- | --- |
| KMP algorithm | Naive algorithm |
| It’s efficient due to the use of the LPS table (pre-processing), in order to avoid the additional effort. | It is inefficient, because of the matches that aren’t stored for further use, which leads to extra work. |
| It requires additional space and pre-processing. | It doesn’t need neither extra space, nor the pre-processing phase. |
| There is no best or worst case, the complexity is standard, O(n+m). | There is a best and worst case, the complexity varies from case to case. |
| Due to it’s incomprehensibility, the algorithm cannot be used by the ones who don’t understand it. | It presents an ease for its understanding. |

Efficiency of the Knut-Morris-Pratt algorithm – multiple tests

Even if the KMP algorithm has a linear complexity, there are cases where the naive approach works a little bit faster.

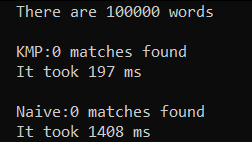
Let’s say that we have a randomly generated text using <https://www.randomtextgenerator.com> (and then we multiply the text to get 1.500.000 words for a larger request from the two algorithms to simulate a demanding use scenario) .

In this case, we’ll search for the word “Can”. The results can be seen down:



As we can see, the time that KMP spent for finding all the occurences is almost twice longer compared to naive solution.

Now let’s take the scenario where we have multiple occurences where the strings match in all but the last letter. We’ll take a string that is composed from 10 identical characters and one that different, at the end. This will help us show the benefits of KMP and the worst case of the naive approach. For a 100.000 text word filled with the string mentioned above, we have these results:



As we can see, the KMP is more 7 times faster and this factor is increased by the length of the text and also the length of the string. The higher speed of KMP is due to its precomputing table that allows an efficient search.

Considering these results, KMP is more stable for every kind of situation, the reason using it being to avoid the bad cases of naive approach and the mismatches.

Conclusion

* The KMP string matching algorithm can be used in various places like - plagiarism detection, digital forensics, spelling checkers etc.
* The naive string matching algorithm would either use a sliding window, or a two pointer approach which would result in extra comparisons. The time complexity for the naive algorithm would be O(mn).
* Components of the KMP algorithm:
* Prefix
* Suffix
* LPS table : Table for detecting the Longest Proper Prefix that is also a Suffix
* In this algorithm, we have 2 pointers and we work on the LPS table and the string. We compare string[i] and pattern[j]. There are 3 actions that could happen on an iteration of the while loop:
* String match, increment i and j
* String mismatch, but j > 0, so move j to LPS[j - 1] and leave i as it is and compare
* String mismatch, but j = 0, so increment i and compare
* Time complexity of the complete algorithm is O(m + n).

BIBLIOGRAPHY

1. *Bibliography of Donald Knuth*, online resource: <https://www.britannica.com/biography/Donald-Knuth>, accessed on 20.12.2022.
2. Cormen T.H., *Introduction to Algorithms*, 3rd Edition (The MIT Press) 3rd Edition, 2009, pp. 1002.
3. *String-searching algorithm*, online resource: <https://en.wikipedia.org/wiki/String-searching_algorithm> , accessed on 20.12.2022.
4. Charras C., Lecroq T., *Handbook of Exact String-Matching Algorithms*, online resource: <https://www-igm.univ-mlv.fr/~lecroq/string/string.pdf>, accessed on 21.12.2022.
5. Knuth D. E., Morris J. H., Pratt V. R., *Fast pattern matching in strings*, SIAM Journal on Computing, 1977.
6. Pedamkar Priya, *KMP Algorithm*, <https://www.educba.com/kmp-algorithm/> , accesed on 31.01.2023.
7. Data structures, *KMP Algorithm*, online resource: <https://www.scaler.com/topics/data-structures/kmp-algorithm/>, accessed on 21.12.2022.
8. COOK S. A., *Linear time simulation of deterministic two-way pushdown automata*, Information Processing 71, North-Holland, Amsterdam, 1972, pp. 75-80.

1. *Bibliography of Donald Knuth*, online resource: <https://www.britannica.com/biography/Donald-Knuth>, accessed on 20.12.2022. [↑](#footnote-ref-1)
2. Cormen T.H., *Introduction to Algorithms*, 3rd Edition (The MIT Press) 3rd Edition, 2009, pp. 1002. [↑](#footnote-ref-2)
3. *String-searching algorithm*, online resource: <https://en.wikipedia.org/wiki/String-searching_algorithm> , accessed on 20.12.2022 [↑](#footnote-ref-3)
4. Charras C., Lecroq T., *Handbook of Exact String-Matching Algorithms*, online resource: <https://www-igm.univ-mlv.fr/~lecroq/string/string.pdf>, accessed on 21.12.2022. [↑](#footnote-ref-4)
5. Knuth D. E., Morris J. H., Pratt V. R., *Fast pattern matching in strings*, SIAM Journal on Computing, 1977 [↑](#footnote-ref-5)
6. Pedamkar Priya, *KMP Algorithm*, <https://www.educba.com/kmp-algorithm/> , accesed on 31.01.2023. [↑](#footnote-ref-6)
7. Data structures, *KMP Algorithm*, online resource: <https://www.scaler.com/topics/data-structures/kmp-algorithm/>, accessed on 21.12.2022. [↑](#footnote-ref-7)