FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report on the practical task No. 8
“Practical analysis of advanced algorithms”

Performed by	Pavel Belenko, J4132C
Accepted by	Dr Petr Chunaev

St. Petersburg
2021

Goal:

Practical analysis of advanced algorithms.

Formulation of the problem:

I. Choose two algorithms that are not considered in the course from the proposed sections of the book: “Introduction to Algorithms”, 2009, by Thomas H., Cormen Charles E., Leiserson Ronald L. Rivest Clifford Stein.

In this paper, two algorithms for solving a string-matching problem (module VII, chapter 32 of the book) will be considered: the Rabin-Karp algorithm and the Knuth-Morris-Pratt algorithm.

II. Analyze the chosen algorithms in terms of time and space complexity, design technique used, etc. Implement the algorithms and produce several experiments. Analyze the results.

Brief theoretical part:

In computer science, string-searching algorithms, sometimes called 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.

A basic example of string searching is when the pattern and the searched text are arrays of elements of an alphabet (finite set) Σ. Σ may be a human language alphabet, for example, the letters A through Z and other applications may use a binary alphabet (Σ = {0,1}) etc.

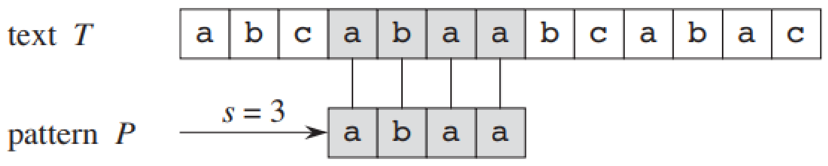


Figure 1 - An example of the string-matching problem from the book

To solve the string-matching problem means to find in text all such positions s where the pattern in question occurs. For example, in Figure 1, where there is only one repetition of the pattern in the line, $s = 3$.

Let's talk in more detail about the selected algorithms. The first of them, the **Rabin-Karp algorithm**, is a string search algorithm that uses hashing to find the exact match of the pattern string in the text. It uses polynomial hash function, which has the following form:

$$hash(str) = (s_0 + s_1 d + s_2 d^2 + \dots + s_n d^n) \; mod \; q,$$

where each s_i is an ordinal number of the element in the alphabet; d is an arbitrary number larger than the size of the alphabet, and q is a sufficiently large module (generally speaking, not necessarily simple).

The second one under consideration, the **Knuth-Morris-Pratt algorithm** (KMP algorithm) is the most efficient algorithm for searching for a substring in a string. The running time of the algorithm depends linearly on the amount of input data, that is, it is impossible to develop an asymptotically more efficient algorithm. The algorithm is based on the use of a prefix function, which is used to search for occurrences of a template in a string.

A prefix function is a function from a string that, for each i-th element of the string, shows the longest length of the substring suffix that does not match the entire string from 0 to the i-th element inclusive, coinciding with its prefix. A prefix is a substring starting from the beginning of a string. And a suffix is a substring ending at the end of a string. Mathematically, the definition of a prefix function can be written as follows:

$$\pi[i] = \max_{k=0 \dots i} \{k : s[0 \, .. \, k-1] = s[i-k+1 \, .. \, i]\}$$

	Algorithm		**Time complexity**	
		Preprocessing	Average	Worst
1	*Brute Force algorithm*	-	$O(p \cdot (t-p))$	$O(t^2)$
2	*Rabin-Karp matcher*	$O(p)$	$O(p+t)$	$O(pt)$
3	*Knuth-Morris-Pratt matcher*	$O(p)$	$O(p+t)$	$O(p+t)$

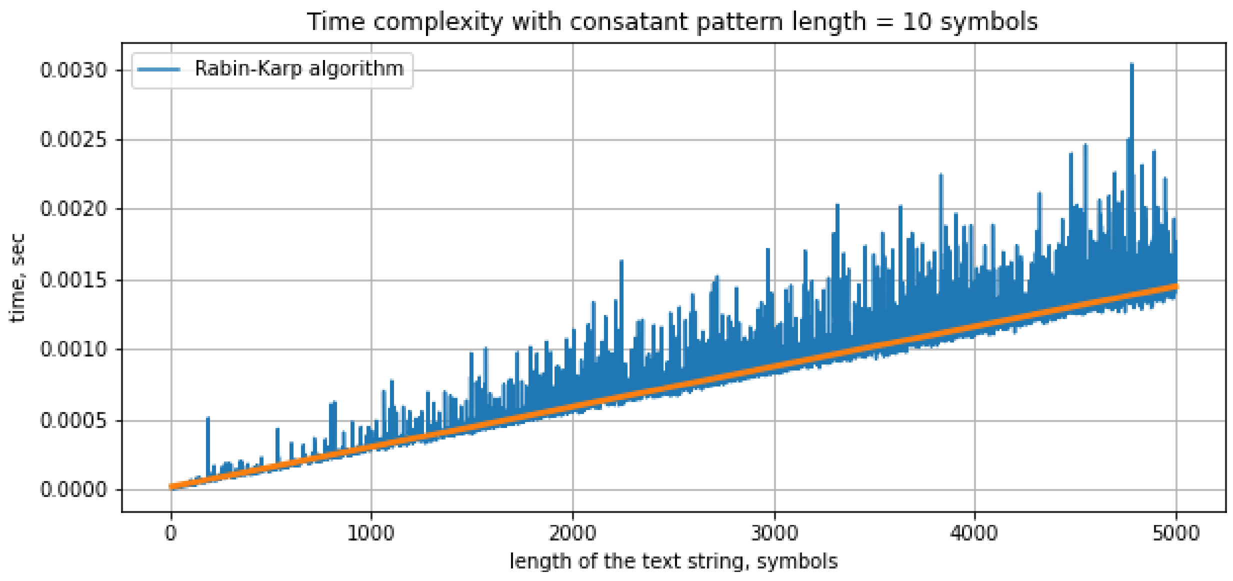
Table 1 - The string-matching algorithms and their preprocessing and matching times (p = length(pattern), t = length(text))

Results:

All calculations were performed in the Jupiter lab development environment. NumPy, time and matplotlib modules were used in the process.

To test the theoretical time complexity of the algorithms, the following experiments were proposed: Figure 1 - algorithms work at a constant pattern length. Figure 2 - algorithms work at a constant text length.

To reduce the influence of performance factors beyond our control, the result was calculated as the average time in 5 repetitions of the algorithm for each specific string length.



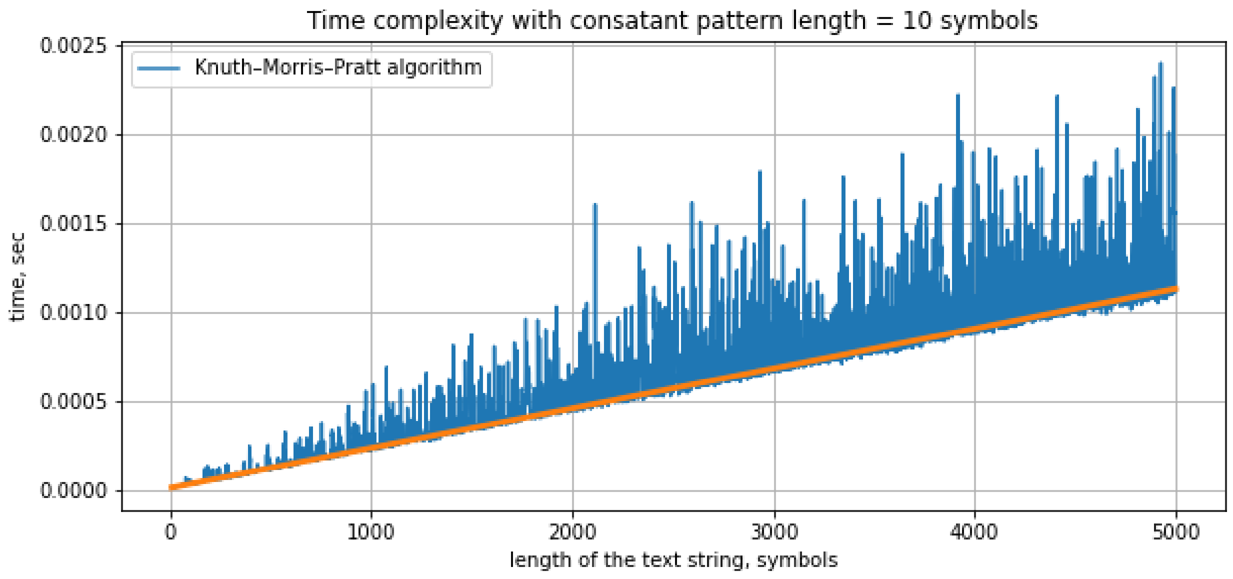
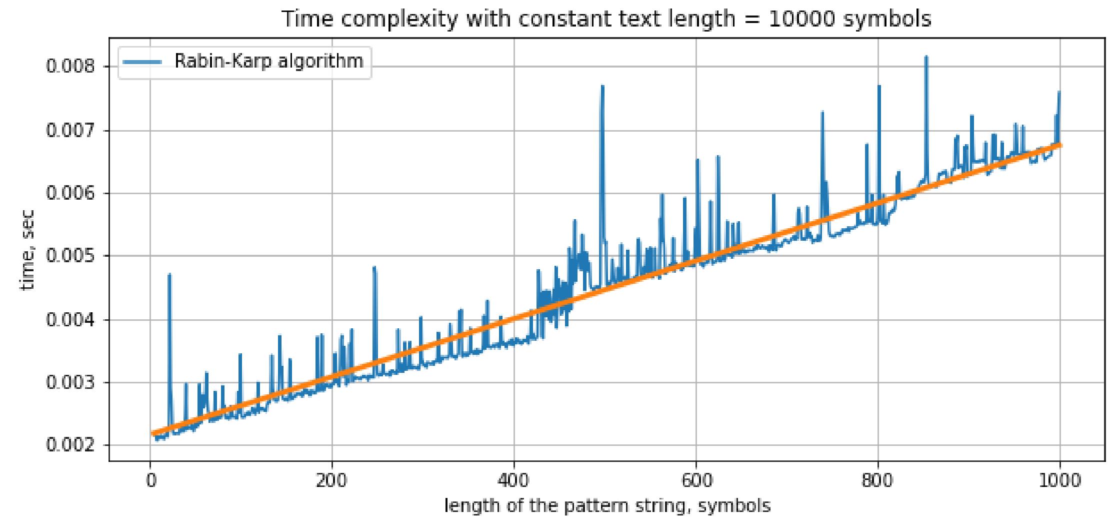


Figure 2 – Time complexity of the algorithms depends on the length of the text string (pattern length is constant)



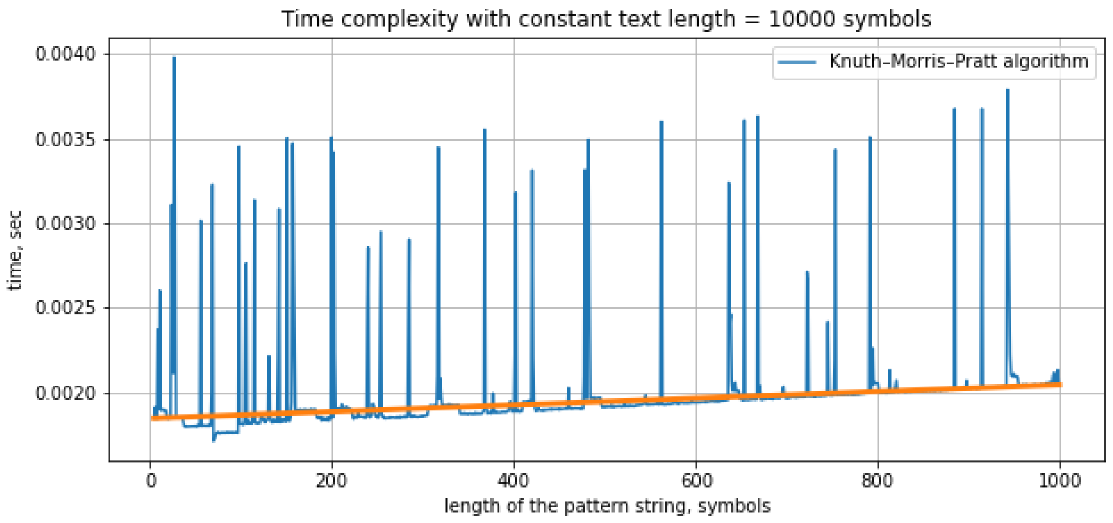


Figure 3 – Time complexity of the algorithms depends on the length of the pattern string (text length is constant)

Conclusions:

In this paper, two algorithms for searching for a substring in a string were implemented. The results obtained are in good agreement with the theory: when fixing one of the variables (the length of the text or the length of the pattern), the time complexity becomes equal to O(n)

Some curvature of the graphs can be explained by the peculiarities of generating text data to test the algorithm, as well as the fact that other applications were running on the computer in parallel with the calculations.

Appendix:

github.com/belpablo/Algorithms-21/blob/main/Lab_8/Lab_8_redo.ipynb