FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report
on the practical task No. 1
“Experimental time complexity analysis”

Performed by Pavel Belenko, J4132c

Accepted by Dr Petr Chunaev

St. Petersburg
2021

Goal: Experimental study of the time complexity of different algorithms.

Formulation of the problem:

Time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken, and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.

Here are examples of theoretical time complexity of some algorithms, where $\boldsymbol{v} = [v_1, v_2, v_3, \dots v_n]$ is an n-dimensional random vector of non-negative integers, A and B are $n \times n$ sized matrices with random non-negative elements:

Algorithm		Time complexity		
		Best	Average	Worst
1	$f(v) = const$		O(1)	
2	$f(v) = \sum_{i=1}^{n} v_i$		O(n)	
3	$f(v) = \prod_{i=1}^{n} v_i$		$O(n^{\log_2 3})$	
4	$P(v,x) = \sum_{i=1}^{n} v_i x^{i-1}$		O(n)	
5	$P(v,x) = = v_1 + x(v_2 + \cdots)$		O(n)	
6	Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
7	Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
8	Timsort	O(n)	$O(n \log n)$	$O(n \log n)$
9	$(A \times B)_{i,j} = = \sum_{k=1}^{n} a_{i,k} b_{k,j}$		$O(n^3)$	

Table 1 – Theoretical time complexity for algorithms 1-9 that need to be implemented in the task

The problem of the task is to experimentally determine the time complexity for these 9 algorithms and compare it with the theoretical one.

Brief theoretical part:

In most cases, in this task we work with vectors. **Vector** (one-dimensional array) is a static data structure with a fixed number of elements of the same type. Each element of the vector has unique number within this vector. Reference to a vector element is performed by the vector name and the number of the required element. In the memory vector is represented by adjacent cells with elements. The address of each element in the vector can be calculated if we know where the zero element and how many bits is each element takes.

Array (multi-dimensional array), which we use, talking about matrices, is a static data structure storing a set of values of the same type, identified by an index or a set of indices. Two-dimensional array is a vector where each element is a vector of the same length. This is a connection between arrays and vectors. There are two variants how arrays can be represented in the memory. The first is to arrange rows of the array in memory one after another. In this case, we are a little bit inefficient in memory usage, but we win in the speed of indexing and initialization. The second one is to store pointers to other arrays in the vector. In this case, we get a slower indexing and win in memory usage, but rows of such array (vector) may have different length.

Considering the algorithms that are used in this practical problem, in paragraphs 1-5 and 9 from Table 1, we use only a simple loop with some mathematical operations applied to each of the elements of the vector. Further, in paragraphs 6 - 8 there are 3 different sorting methods that should be considered separately:

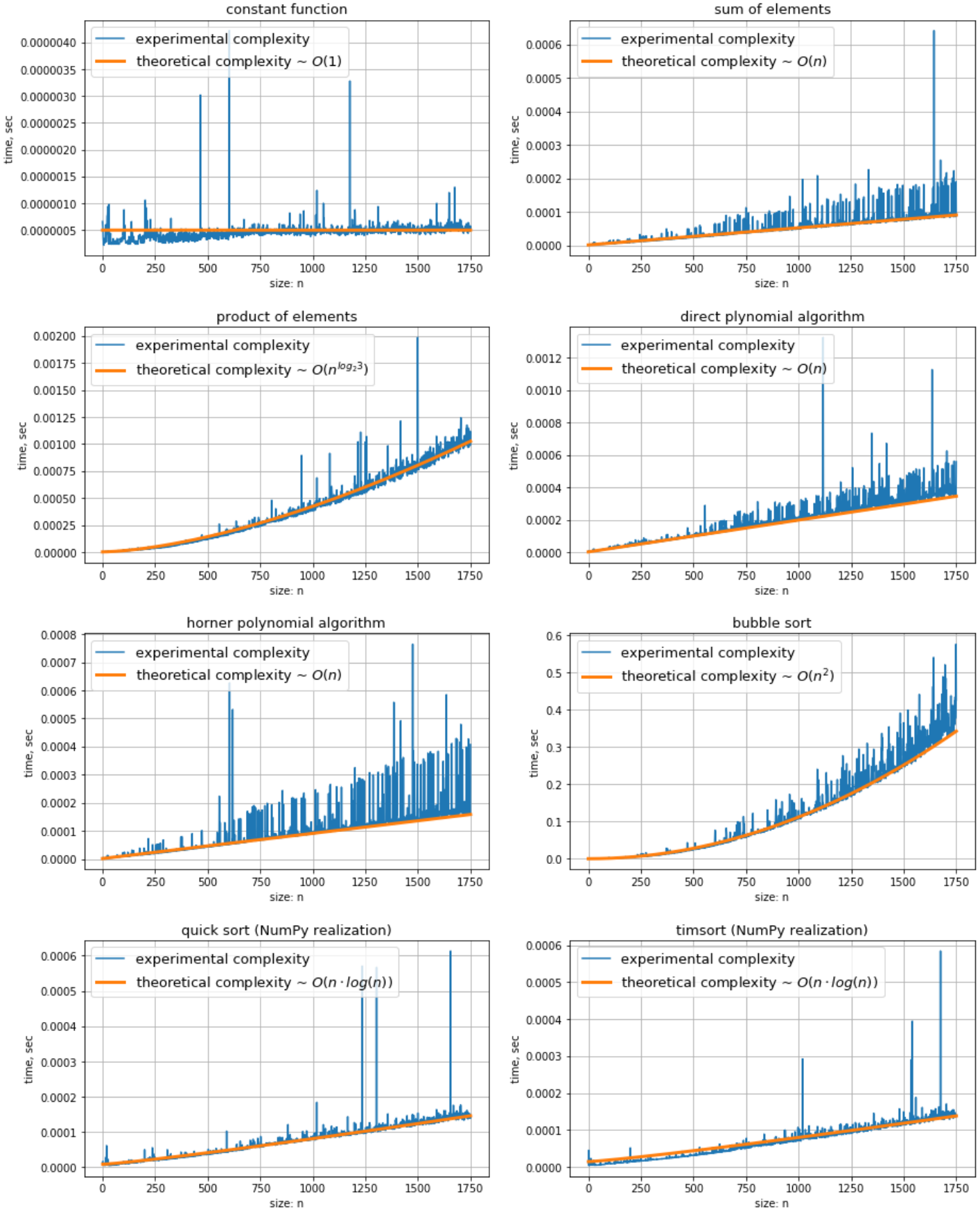
1) **Bubble sort** is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.
   This simple algorithm performs poorly in real world use and is used primarily as an educational tool.

2) **Quicksort** is a “divide and conquer” sorting in the style “merge sorting”. The main idea is to find pivot in the array to compare with the rest of the elements, then shift the elements so that all the parts before the pivot are smaller than it, and the elements after the pivot are larger than it. Then, this process is recursively applied to each of the parts

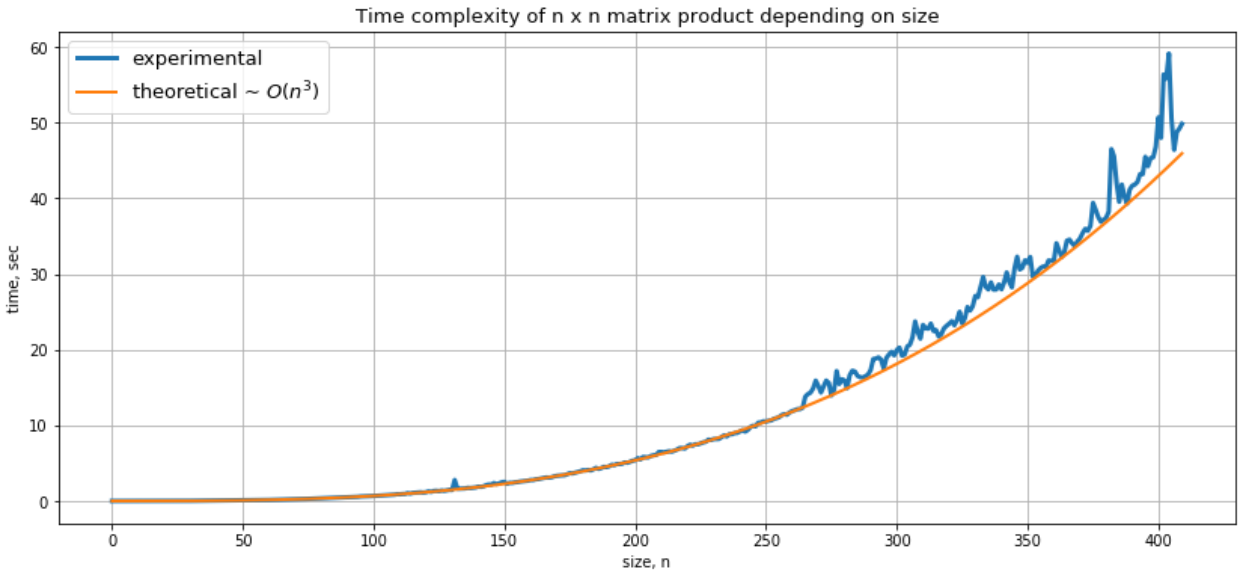
3) **Timsort** is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. The main idea is that, according to the specific algorithm, the input array is divided into subarrays. Then, each subarray is sorted by insertion sort. Finally, the sorted subarrays are collected into a single array using modified merge sort.

Results:

All calculations were performed in the Jupiter lab development environment. NumPy, random, time and matplotlib modules were used in the process. For each n from 1 to 2000, the average computer execution time (using timestamps) of programs implementing the algorithms and functions listed below was measured for five runs.



Picture 1 – Time complexity depending on size of vector for each described algorithm in two ways – experimental and theoretical



Picture 2 – Time complexity of n x n matrix product depending on size in two ways – experimental and theoretical

Conclusions:

As a result of the work, the time complexities of 9 different algorithms were investigated and compared with the theoretical results. Even though there are many deviations in the modeling, the results obtained are well comparable with the theoretical ones. In fact, fluctuations in the graphs may be associated with an additional load on the RAM and the central processor of the computer during calculations.

It should also be mentioned that, speaking of polynomials, the Horner's scheme has the same time complexity as direct calculations of the polynomial, but it provides 2 times faster calculations.

Appendix:

To view the code in GitHub repository, follow the link:

https://github.com/belpablo/Algorithms-21