FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
ITMO UNIVERSITY

Report

on the practical task No. 1

“Experimental time complexity analysis”

Performed by

Dmitriy Prusskiy

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2021

Goal

Experimental study of the time complexity of different algorithms.

Formulation of the problem

For each n from 1 to 2000, measure the average computer execution time (using timestamps) of programs implementing the algorithms and functions below for five runs. Plot the data obtained showing the average execution time as a function of n. Conduct the theoretical analysis of the time complexity of the algorithms in question and compare the empirical and theoretical time complexities.

I. Generate an n-dimensional random vector $\boldsymbol{v} = [v_1, v_2, \ldots, v_n]$ with non-negative elements. For $\boldsymbol{v}$, implement the following calculations and algorithms:

1) $f(\boldsymbol{v}) = const$ (constant function);
2) $f(\boldsymbol{v}) = \sum_{k=1}^{n} v_k$ (the sum of elements);
3) $f(\boldsymbol{v}) = \prod_{k=1}^{n} v_k$ (the product of elements);
4) supposing that the elements of $\boldsymbol{v}$ are the coefficients of a polynomial P of degree $n - 1$, calculate the value $P(1.5)$ by a direct calculation of $P(x) = \sum_{k=1}^{n} v_k x^{k-1}$ (i.e. evaluating each term one by one) and by Horner's method by representing the polynomial as $P(x) = v_1 + x(v_2 + x(v_3 + \ldots))$;

5) Bubble Sort of the elements of $\boldsymbol{v}$;
6) Quick Sort of the elements of $\boldsymbol{v}$;
7) Timsort of the elements of $\boldsymbol{v}$.

II. Generate random matrices A and B of size $n \times n$ with non-negative elements. Find the usual matrix product for A and B.

III. Describe the data structures and design techniques used within the algorithms.

Brief theoretical part

The **time complexity** of an algorithm is the number of elementary operations performed while the algorithm is running. To express the time complexity of an algorithm, we use something called the "Big O notation". The Big O notation is a language we use to describe the time complexity of an algorithm. Big O notation expresses the run time of an algorithm in terms of how quickly it grows relative to the input (this input is called "n").

Constant function is a function whose result is the same for any input. Its time complexity is $O(1)$.

The functions "**sum of elements**" and "**product of elements**" perform one elementary operation on each element of the vector. Their time complexity is $O(n)$.

The function calculating the value of a polynomial of degree n-1 in a **direct way** needs to calculate all powers of x from 0 to n-1 and multiply them by the elements of the vector v. For best performance, it is possible to keep the power of x at each step and use it in the next step. In this way the function has time complexity $O(n)$.

A function that calculates the value of a polynomial of degree n-1 using **Horner's method** performs one addition and two multiplications at each step. Thus, the function has time complexity $O(n)$.

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. Bubble sort algorithm utilizes two loops: an outer loop to iterate over each element in the input list, and an inner loop to iterate, compare and exchange a pair of values in the list. The inner loop takes n-1 iterations while the outer loop takes n iterations. Hence, Bubble sort has$O(n^2)$time complexity.

Quicksort is a type of divide and conquer algorithm for sorting an array, based on a partitioning routine. Applied to a range of at least two elements, partitioning produces a division into two consecutive non empty sub-ranges, in such a way that no element of the first sub-range is greater than any element of the second sub-range. After applying this partition, quicksort then recursively sorts the sub-ranges, possibly after excluding from them an element at the point of division that is at this point known to be already in its final location. Quicksort average time complexity is $O(nlogn)$.

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. Timsort was designed to take advantage of runs of consecutive ordered elements that already exist in most real-world data, natural runs. It iterates over the data collecting elements into runs and simultaneously putting those runs in a stack. Whenever the runs on the top of the stack match a merge criterion, they are merged. This goes on until all data is traversed; then, all runs are merged two at a time and only one sorted run remains. The advantage of merging ordered runs instead of merging fixed size sub-lists (as done by traditional mergesort) is that it decreases the total number of comparisons needed to sort the entire list. Timsort average time complexity is $O(nlogn)$.

Matrix multiplication is an operation on two matrices, the result of which is a matrix whose elements are calculated as follows:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Thus, in the case of square matrices, the computation of each element of the resulting matrix has time complexity $O(n)$. And matrix multiplication has time complexity $O(n^3)$.

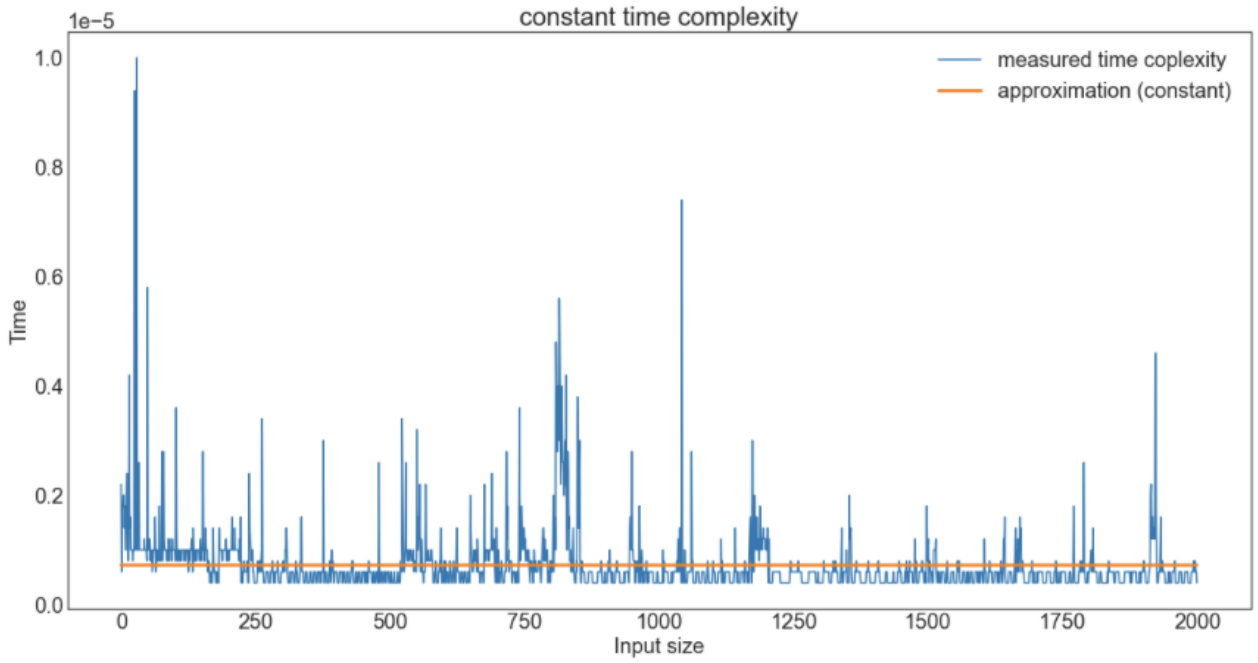
Results

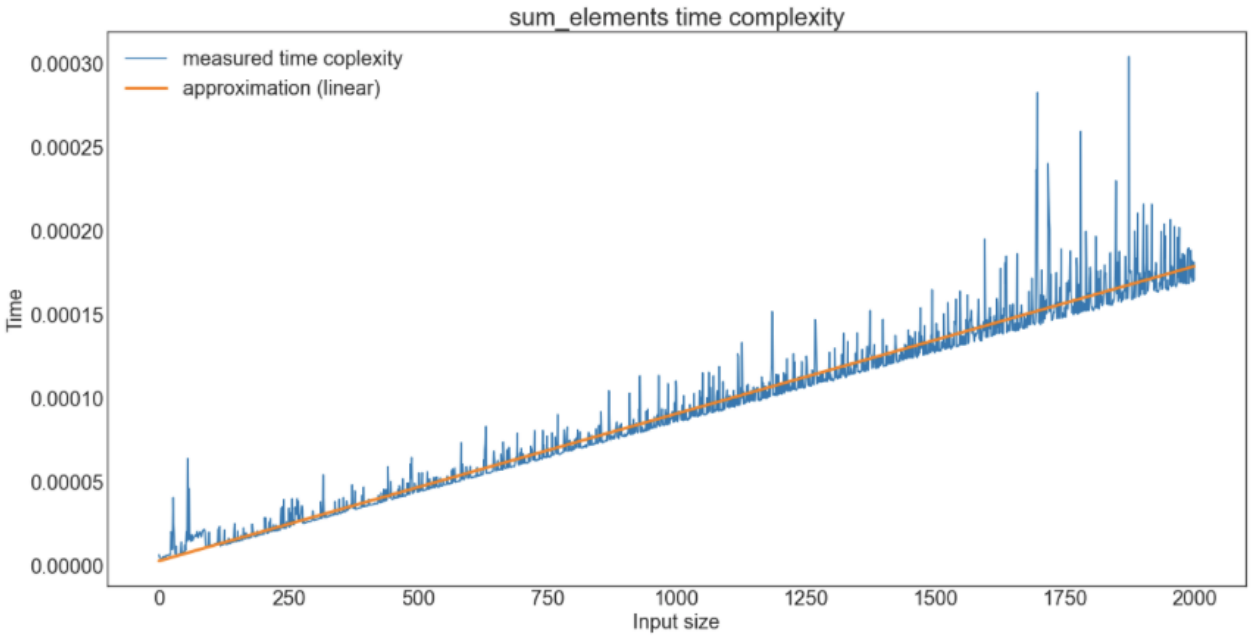
The results were obtained using Python 3.8.5 on a computer with 2.3 GHz CPU frequency.

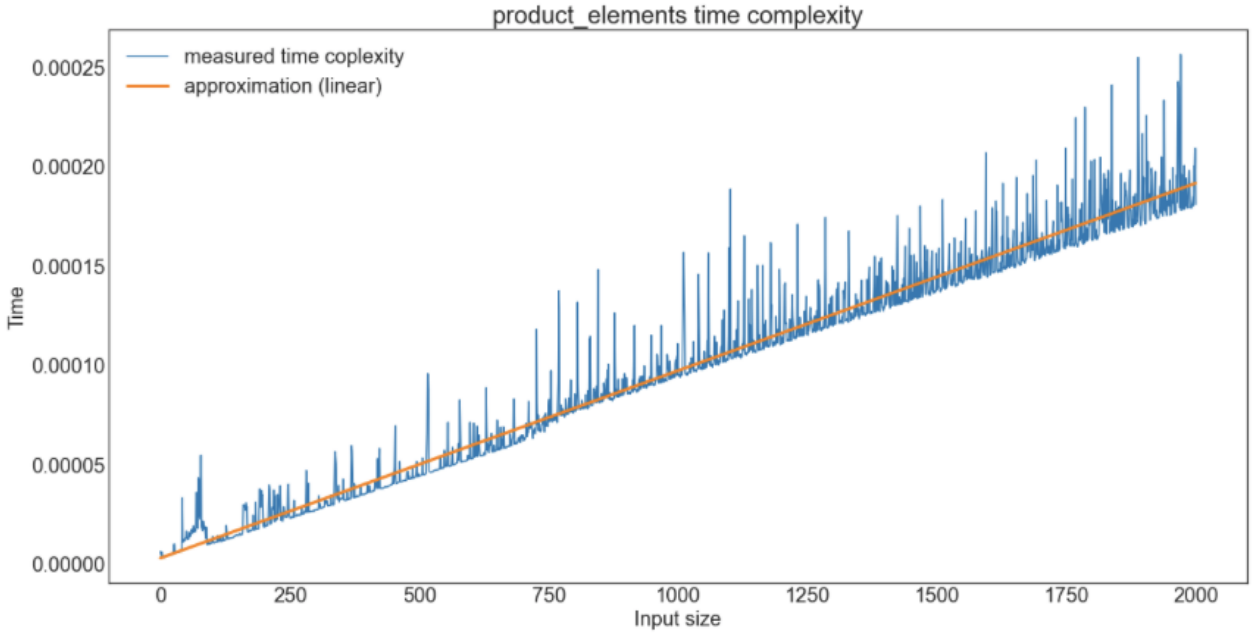
For each n from 1 to 2000, was measured the average computer execution time of programs implementing the algorithms and functions described in the previous part. Obtained results were plotted with approximation curves, representing theoretical time complexities.

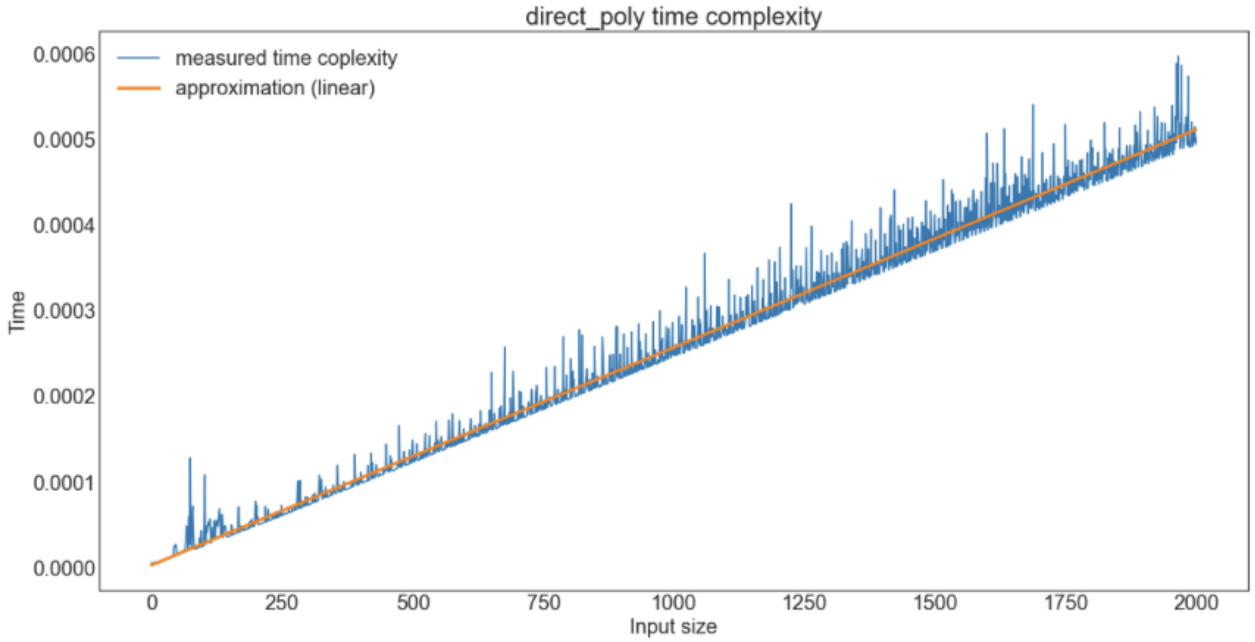
Theoretical approximations were obtained by using *numpy.polynomial.polynomial.polyfit* method. The degrees of the polynomials for the approximation were set to 1 (except for the constant function, where the degree was set to 0). As predictors for all functions except sorting methods were used x. For Bubble sort was used x^2 as a predictor. For Quicksort and Timsort were used $x * ln(x)$. And for matrix multiplication - x^3.

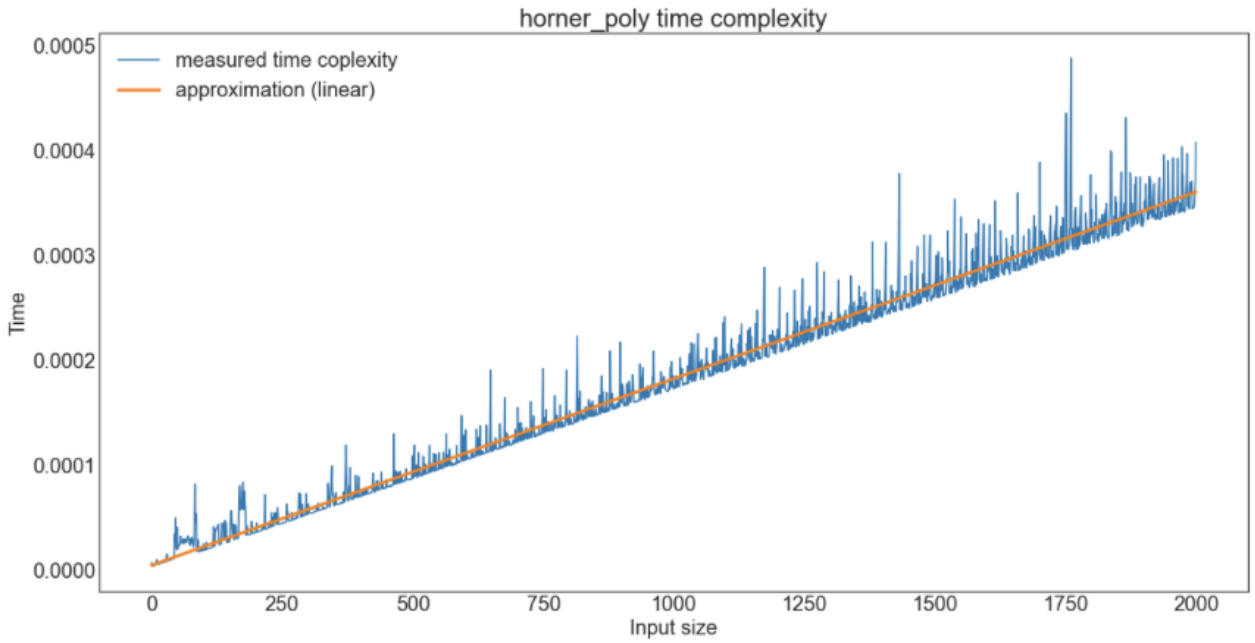
Below are graphs with the results of the study.

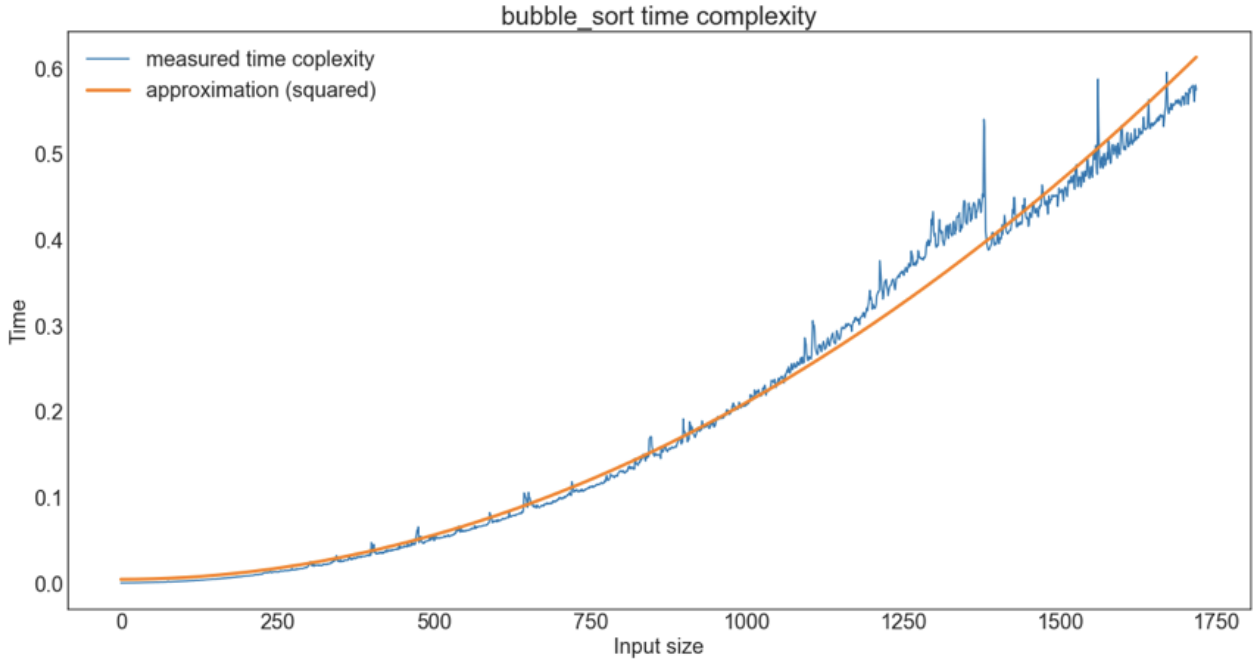



sum_elements time complexity
measured time coplexity
approximation (linear)
Time
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0
250
500
750
1000
1250
1500
1750
2000
Input size

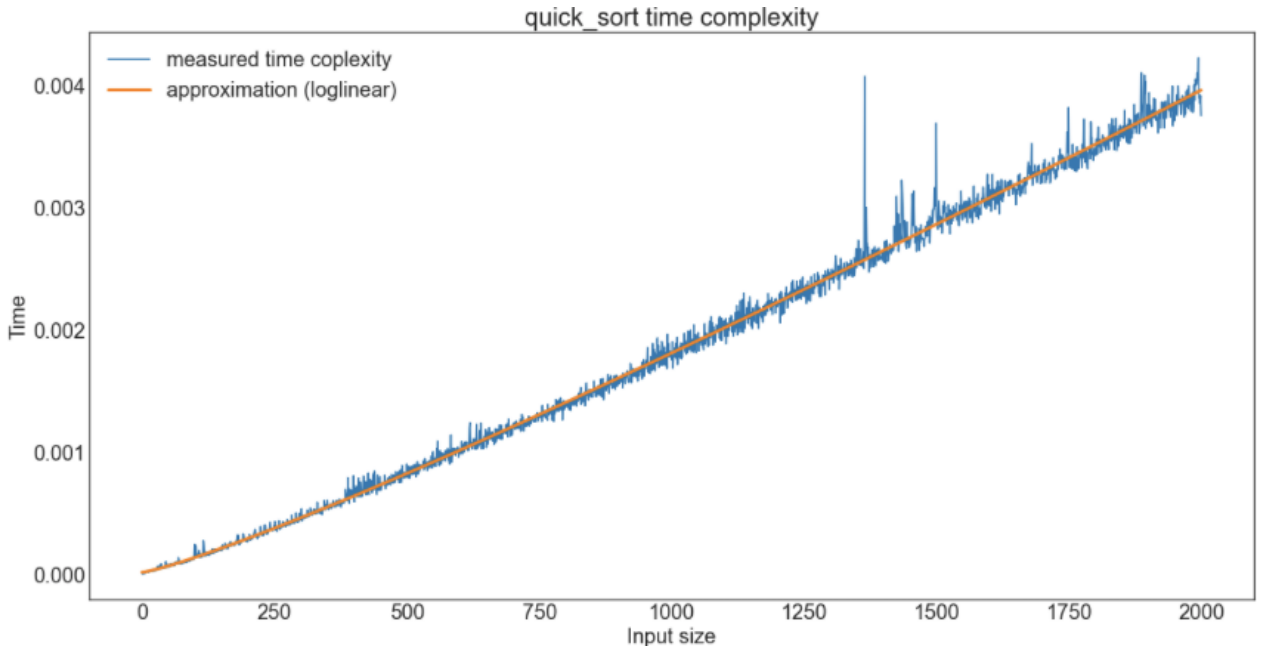

product_elements time complexity
measured time coplexity
approximation (linear)
Time
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0
250
500
750
1000
1250
1500
1750
2000
Input size

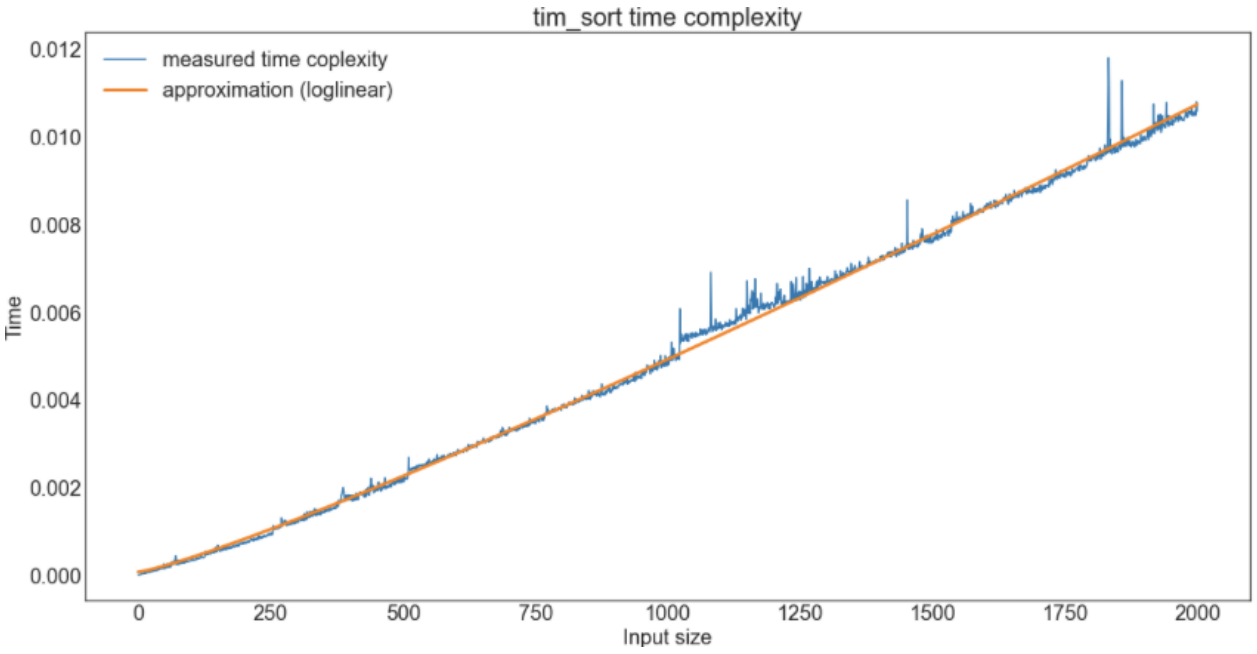

direct_poly time complexity
measured time coplexity
approximation (linear)
Time
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
250
500
750
1000
1250
1500
1750
2000
Input size

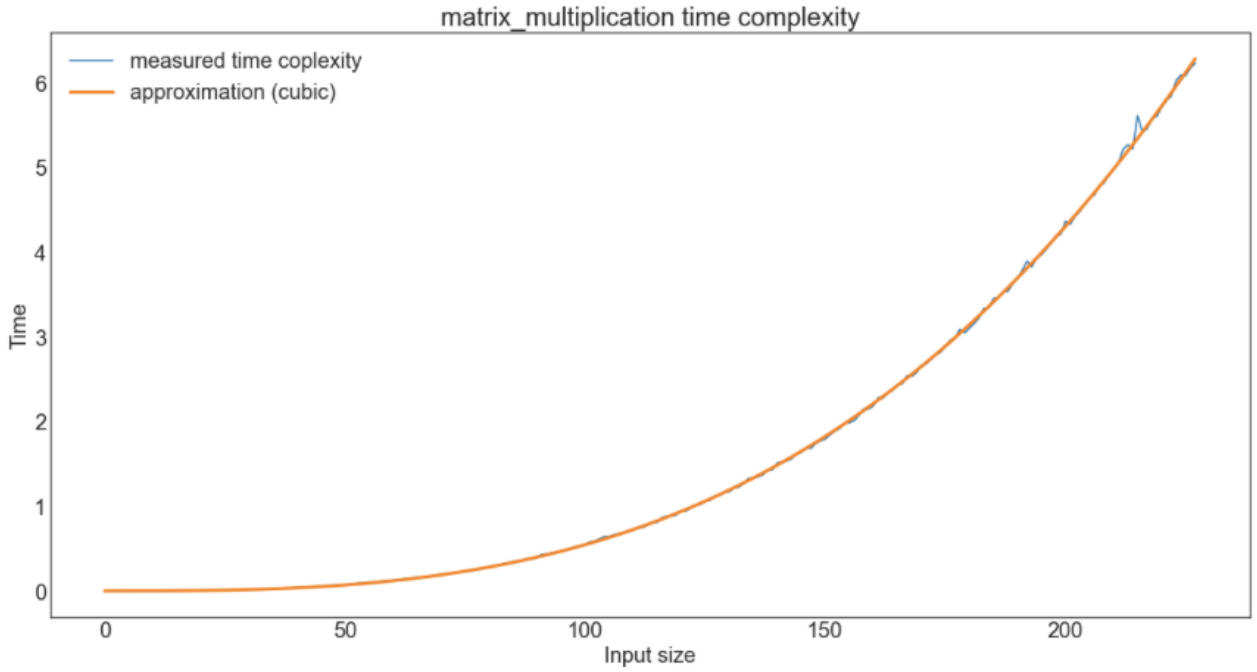

horner_poly time complexity
measured time coplexity
approximation (linear)
Time
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
250
500
750
1000
1250
1500
1750
2000
Input size



Bubble sort takes a lot of time to run, so it exceeded the time limit (30 minutes) on the 1718-th step.







Matrix multiplication takes a lot of time to run, so it exceeded the time limit (30 minutes) on the 228-th step.

Conclusions

The goal of this study was to measure the time complexity of different algorithms and compare obtained results with theoretical time complexities for these algorithms. Analysis of the graphs presented in the Results section shows that the experimental time complexities converge with the

theoretical ones.

Appendix

Source code can be found at https://github.com/T1MAX/itmo_algorithms/tree/main/task_1