FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report

on the practical task No. 1

Experimental time complexity analysis

Performed by

Dmitry Grigorev, Maximilian Golovach

J4133c

Accepted by

Dr Petr Chunaev

St. Petersburg

2022

1. Goal of the work

The goal of this work is to examine time complexity of a number of algorithms and functions empirically and to compare it with their theoretical estimates.

2. Formulation of the problem

For each $n = 1, \ldots, 2000$ the following algorithms are considered in this work:

- For arbitrary $v \in \mathbb{R}^n_+$ with non-negative elements we need to:
  - calculate $f(v) = const$,
  - calculate $f(v) = \sum_{k=1}^{n} v_k$,
  - calculate $f(v) = \prod_{k=1}^{n} v_k$,
  - calculate $P(x) = \sum_{k=1}^{n} v_k x^{k-1}$ at $x = 1.5$ directly,
  - calculate $P(x) = \sum_{k=1}^{n} v_k x^{k-1}$ at $x = 1.5$ by Horner's method,
  - do Bubble sort of the elements of v,
  - do Quick sort of the elements of v,
  - do Tim sort of the elements of v;
- For arbitrary $(n \times n)$-matrices A and B with non-negative elements we need to find their usual product.

Finally, we have to describe the data structures and design techniques used within the algorithms.

3. Brief theoretical part

3.1. Algorithms time complexity

First of all, it is worth defining the notion of time complexity.

Definition 1. **Time complexity** is defined as the amount of time taken by an algorithm to run, as a function of the length of the input.

For given algorithm the estimate of its time complexity is calculated by means of counting of the number of elementary operations (such as addition, product and etc.) needed to execute this algorithm. The time of an elementary operation execution is supposed to

be constant. Since the execution time depends on the amount of data n, it is conventionally treated as a function $T: \mathbb{N} \to \mathbb{R}_+$ which describes how execution time $T(n)$ of given algorithm behaves under change of the amount of data n passed to the algorithm.

Often it is hard or impossible to find the exact expression of $T(n)$. Moreover, exact values of $T(n)$ are not as interesting as the behaviour of this function as $n \to \infty$, i.e. its **asymptotical behaviour**. Therefore, instead of consideration of $T(n)$ its O-notation is used.

Definition 2. Function $T(n)$ is said to be $O(t(n))$ if

$$\exists C > 0 \; \exists N \in \mathbb{N}: \; \forall n > N \; T(n) \leq Ct(n),$$

where $t: \mathbb{N} \to \mathbb{R}_+$.

In order to investigate empirically the time complexity of given algorithm, the algorithm is run for each n which varies in fixed range (for instance, from 1 to 2000). The most important assumption in empirical time complexity measurement is that the algorithm is run under constant conditions (for example, it runs on the same computer and no other processes which can influence on execution of algorithm under consideration). As in practice it is hard to guarantee the constant conditions, the algorithm is run several times (say 5 times) for each n and the execution time is averaged.

As soon as execution time measurements are obtained, it is worth illustrating these results by means of drawing a graph or building a table.

3.2. Polynom calculation

There is a variety of algorithms for calculation of polynoms. In this work only direct and Horner's methods are studied.

Direct method calculates arbitrary polynom by definition:

$$P(x) = v_1 + v_2 x + v_3 x^2 + \cdots + v_n x^{n-1},$$

where we have to calculate power of x in every member of P directly. As for Horner's method, it is based on the following polynom representation:

$$P(x) = v_1 + x(v_2 + x(v_3 + \cdots)),$$

i.e. one starts calculation from multiplication of v_n with x, then one adds v_{n-1} to the results, then multiplies it with x and so on.

3.3. Quick sort

Quick sort acts as follows. Initially it picks a pivot element in array and moves it to its natural place accordingly to order, so that all elements smaller than the picked pivot are moved to the left side of the pivot, and all greater elements are moved to the right side. This part of the algorithms is called partitioning. Finally, the algorithm sorts the resulted subarrays on the left and right of the pivot element by applying partitioning function to them and so on until our subarrays will not be a simple case with array that has only 2 elements. The best complexity of algorithm is $O(n\log(n))$, the worst is $O(n^2)$ (see [1]). Here the worst case can happen when pivot element is smallest or biggest element of an array.

3.4. Tim sort

Tim sort is a sorting algorithm that is based on merge sort and insertion sort. Sorted array is divided into blocks, which are called "runs". Those resulted runs are sorted with insertion sort and then are merged like subarrays in the merge sort. If the array size is less or equal to "run" 's size, array is sorted just by using insertion sort. In practice 32 is usually picked as minimal size of "run". To clarify the idea behind picking size of a run as a power of 2, it is worth mentioning that merge function performs well on subarrays with 2^k size, where $k \in \mathbb{N}$. And the idea of tim sort based on the fact that insertion sort performs well for small arrays. The combination of those facts makes algorithm complexity $O(n)$ in the best case and $O(n\log(n))$ in average case what is proved in [2].

4. Results

For each fixed $n \in 1, \ldots, 2000$ all aforementioned algorithms were run five times and corresponding results were averaged. The measurements results are presented below. In addition, curve which fits the measurements data in the sense of least squares (red curves in graphs) is provided. The curve equation is presented in the top left corner of each graph.

4.1. Constant function

In the graph 1 we observe that constant function calculation exhibits constant time complexity. It coincides with theoretical estimate since the calculation requires only one function call and one function value return.

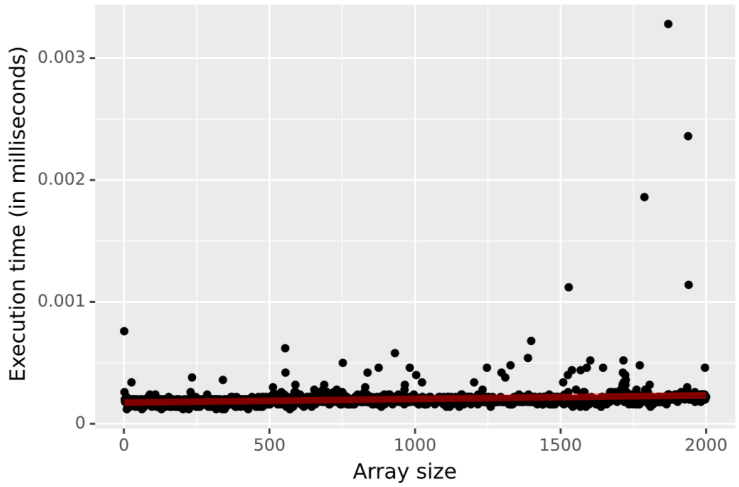

Figure 1: Constant function calculation time complexity measurements.

4.2. Sum of elements

In the graph 2 one can see linear dependence of calculation time versus array size. As summation of n elements requires n additions, these results consistent with theoretical time complexity $O(n)$.

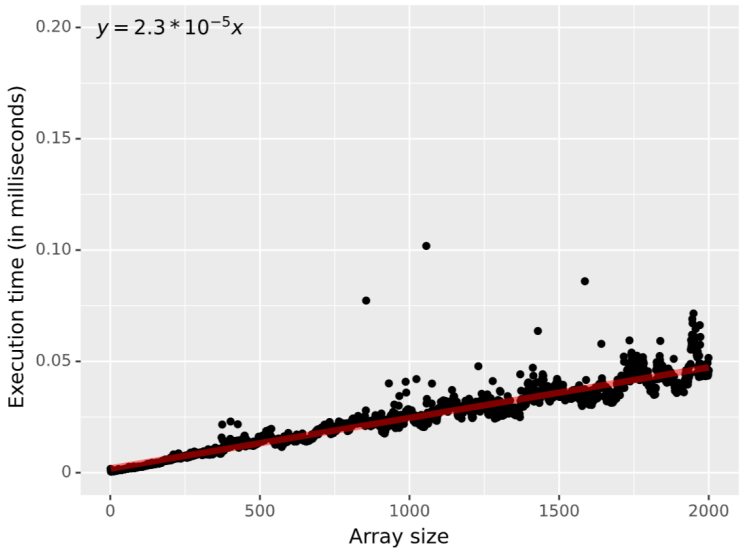

Figure 2: Sum function calculation time complexity measurements.

4.3. Product of elements

The product of n elements requires n multiplications, so it has $O(n)$ time complexity. The graph 3 surely demonstrates this behaviour.

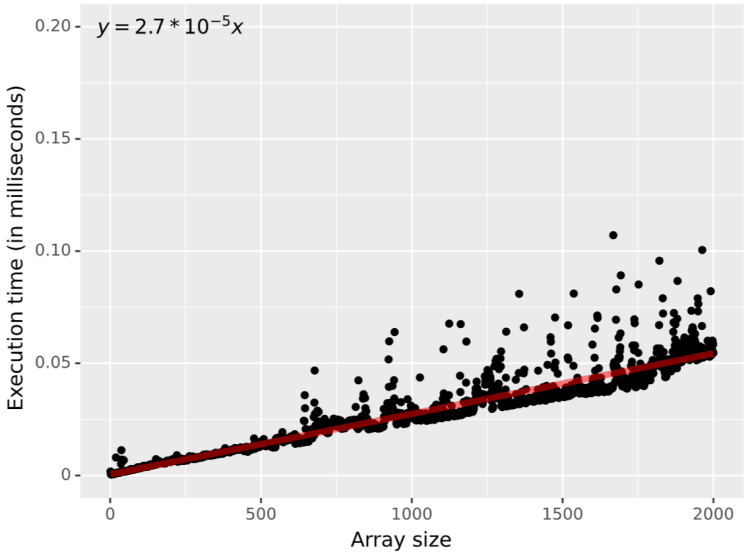

Figure 3: Product function calculation time complexity measurements.

4.4. Polynom calculation

As for the direct method is concerned, corresponding results are presented in the graph 4. It demonstrates quadratic behaviour of time complexity as n increases. It reflects theoretical description since in theory every member in polynom of power $n-1$ requires $O(n)$ multiplications of x on itself, i.e. $O(n^2)$ operations.

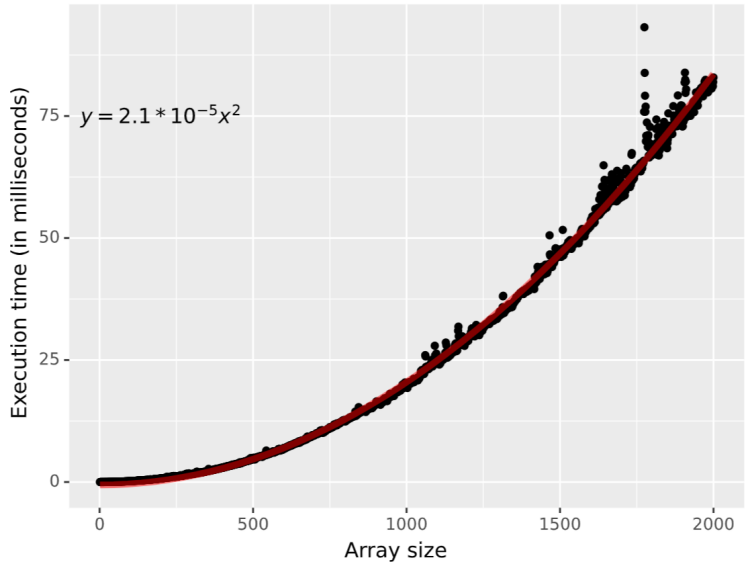

Figure 4: Polynom calculation time complexity measurements. Direct method.

On the contrary, Horner's method is more optimized since it requires $O(n)$ multiplications and $O(n)$ additions. The graph 5 surely exhibits theoretical estimate.

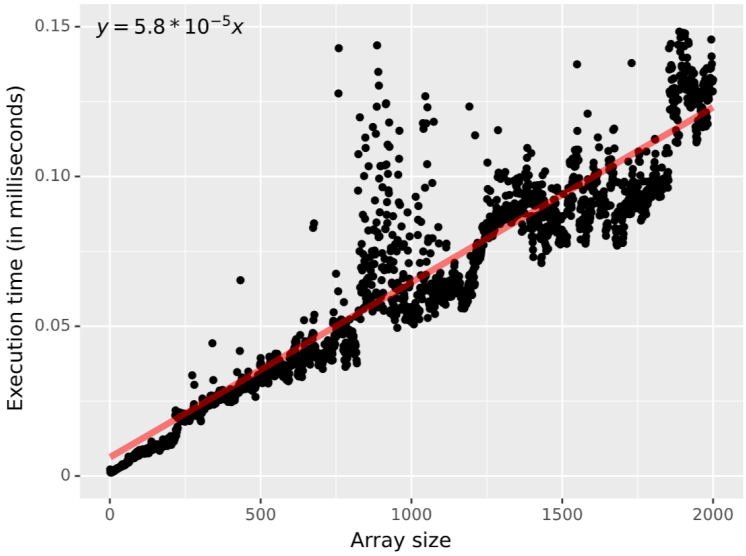

Figure 5: Polynom calculation time complexity measurements. Horner's method.

4.5. Bubble sort

In case of bubble sort, each iteration from n ones requires $O(n)$ comparisons of two adjacent elements, i.e. we have $O(n^2)$ operations. On the graph 6 our calculations present this fact indeed.

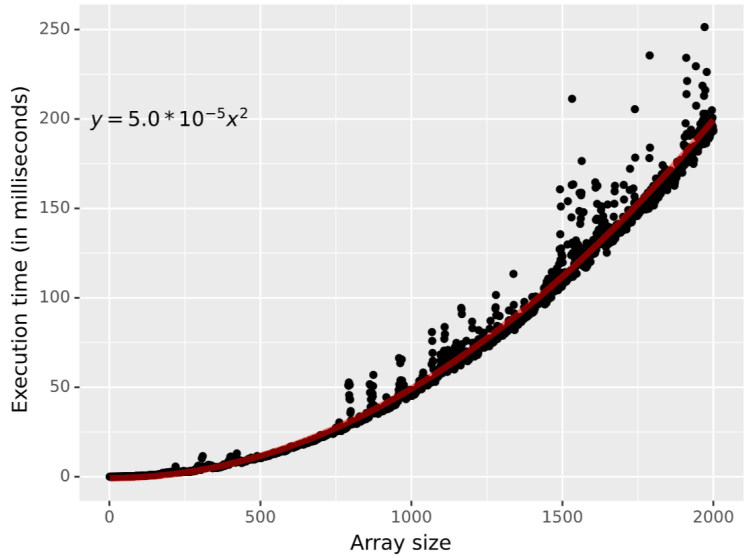

Figure 6: Bubble sort time complexity measurements.

4.6. Quick sort

The theoretical time complexity of quick sort is $O(n \log n)$ in the average case scenario, as has been mentioned in the theory section. Our conducted simulations which results are

depicted in the graph 7 implicitly reflect theoretical behaviour in time.

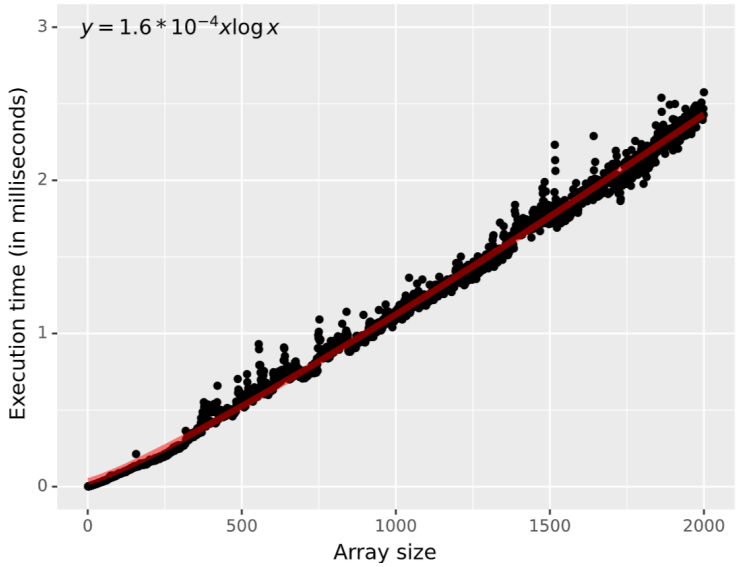

Figure 7: Quick sort time complexity measurements.

4.7. Tim sort

In the figure 8 our measurements of time complexity of Tim sort are presented. As has been discussed in the theory section, Tim sort has $O(n \log n)$ time complexity. Such behaviour is surely observed in the graph so empirical results coincide with theoretical one.

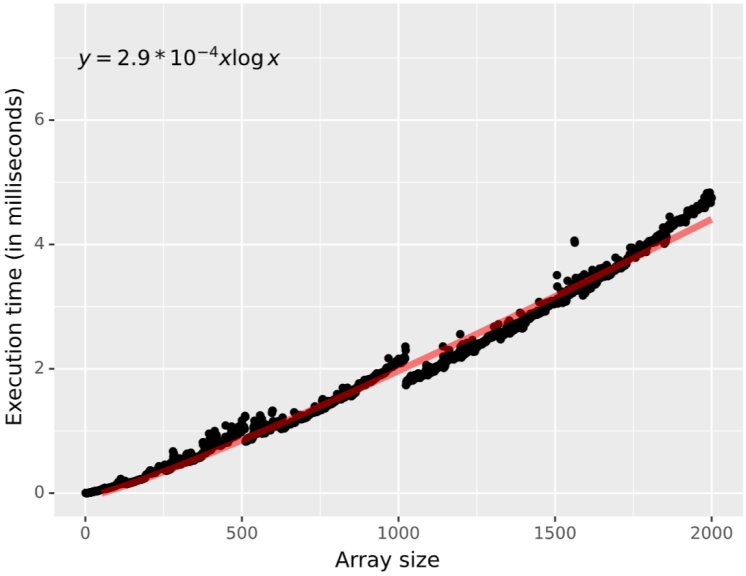

Figure 8: Tim sort time complexity measurements.

4.8. Matrix multiplication

The graph 9 illustrates time complexity measurements for matrices multiplication for every n. Here we observe cubic complexity trend what coincides with theoretical one: every elements from n^2 elements of resulted matrix are calculated by means of $O(n)$ multiplications and $O(n)$ additions. Here we vary n between 1 and 100 since the direct matrix multiplication algorithm requires a lot of time to perform itself.

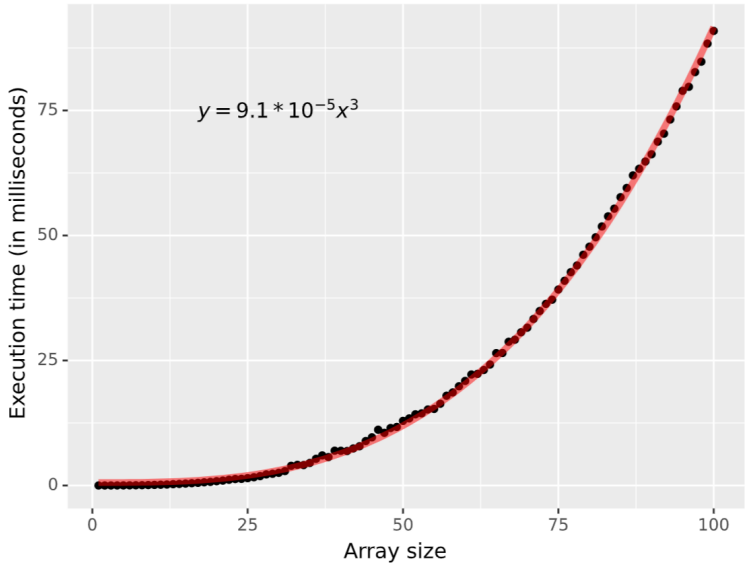

Figure 9: Time complexity measurements for matrix product.

5. Data structures and design techniques used in algorithms

As for polynom calculation, for large n this operation causes overflow issue. Numpy function **numpy.power** can handle it so it does not throw overflow exception as opposed to standard power $**$ operator. This exception ruined our calculation. Since the result of polynom calculations is not as important as its process in our experiments, we implement function **fake_cacl_polynom_direct** which replicates all steps of real direct polynom calculations but in idle manner so it might be claimed that calculations implemented in **fake_cacl_polynom_direct** function are practically identical to standatd direct polynom calculation.

Then, it has to be mentioned that quick sort implementation is iterative and it uses stack data structure in our code.

6. Conclusion

As has been seen from graphs, computations results surely illustrate theoretical behaviour of functions and algorithms in time. Besides, the functions and algorithms were implemented in Python language. Discussion for data structures and design techniques which were used in implementations is provided. The work goals were achieved.

7. Appendix

Algorithms implementation code is provided in [3].

Bibliography

1. Xiang W. Analysis of the Time Complexity of Quick Sort Algorithm // International Conference on Information Management, Innovation Management and Industrial Engineering. — 2011. — 11. — Vol. 1. — P. 408–410.
2. Buss S., Knop A. Strategies for Stable Merge Sorting. — 2018. — online; accessed: `https://arxiv.org/abs/1801.04641`.
3. Grigorev D., Golovach M. Code repository. — `https://github.com/dmitry-grigorev/AlgoAnalysisDevelopment`. — 2022.