FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

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

Performed by
Eugenia Khomenko
J4133c
Accepted by
Dr Petr Chunaev

St. Petersburg 2022

Goal

This work aims for experimental study of the time complexity of different algorithms.

Formulation of the problem

The main task is to measure the average computer execution time of programs implementing the algorithms and functions below for five runs for each *n* from *1* to *2000*. This work must contain the theoretical analysis of the time complexity of the algorithms in question and comparison the empirical and theoretical time complexities. The results of average execution time as a function of n should be visualised as graphs.

Such calculations and algorithms as constant function, the sum and the product of elements, a polynomial function of degree *n-1*, Horner's method, bubble sort, quicksort and timsort are used in first part.

Second part deals with the product of matrices.

Last part requires to describe the data structures and design techniques used within the algorithms.

Brief theoretical part

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. Also an algorithm can appear as a tool for solving a well-specified computational problem.

Usually computer professionals use the method which is the easiest to implement. But at the same time bounded resources of computers should be taken into account because computers are not yet infinitely fast and memory is not free. Efficient in terms of time or space algorithms will help us use these resouces wisely.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. So it is highly required to use a method to compare the solutions in order to judge which one is more optimal. Based on above-mentioned limited resources there are two methods: space complexity and time complexity.

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input.

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. The time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on. This paper is focused only on the time complexity.

Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size. Less common, and usually specified explicitly, is the average-case complexity, which is the average of the time taken on inputs of a given size. In both cases, the time complexity is generally expressed as a function of the size of the input. The time complexity is commonly expressed using big *O* notation.

Algorithmic complexities are classified according to the type of function appearing in the big *O* notation:

- constant time algorithm with time complexity *O(1)*;
- linear time algorithm with time complexity *O(n)*;
- quadratic time algorithm with time complexity $O(n^2)$;
- logarithmic time algorithm with time complexity *O(log n)*;
- linearithmic time algorithm with time complexity *O(n * log n)* etc.

Also such used algorithms as buble sort, quicksort and timsort should be described.

- Buble sort - the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity $O(n^2)$ is quite high. Buble sort example can be seen on image 1.
- Quicksort is divide and conquer algorithm which picks an element as a pivot and partitions the given array around the picked pivot. This algorithm has worst-case $O(n^2)$ time complexity and average time complexity *O(n * log n)*. Quicksort example can be seen on image 3.
- Timsort is a sorting algorithm based on insertion sort and merge sort. It divides the array into blocks known as *run*, sorts those *runs* using insertion sort one by one and then merge those *runs* using the combine function used in merge sort. If the size of the array is less than *run*, then array gets sorted just by using insertion sort. The size of the *run* may vary from 32 to 64 depending upon the size of the array. Timsort example can be seen on image 2.

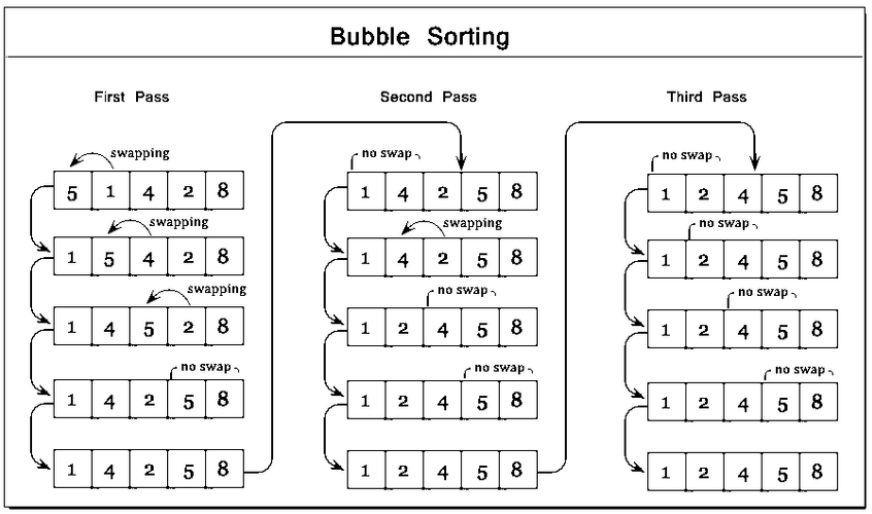


Image 1 - Buble sort example

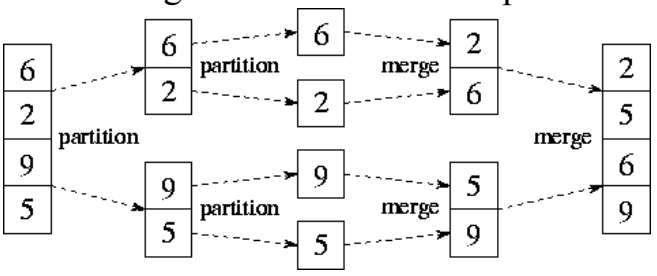


Image 2 - Timsort example

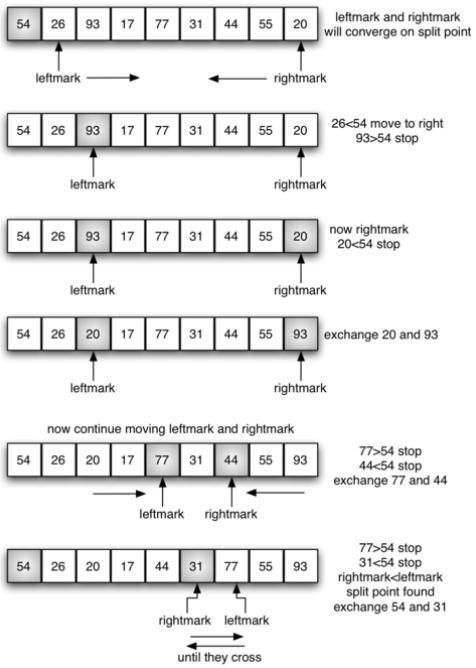


Image 3 - Quicksort example

Results

I. Generated *n*-dimensional random vector *v*: `[4552, 69914, 76111, 27984, 26306,` .

1) Constant function: $f(v) = 1$ with average time complexity *O(1)*. An empirically obtained curve may allow for large deviations from theoretical curves, but on average it is clear that there is a correspondence between experiment and theory.

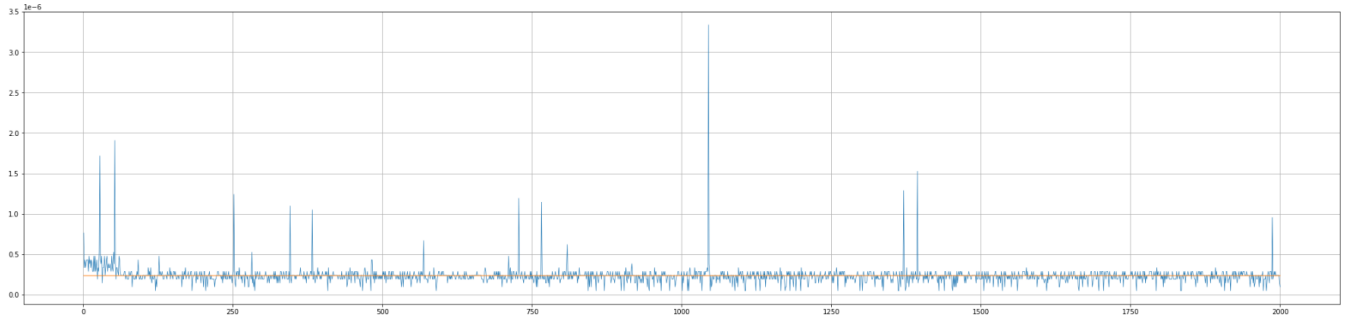
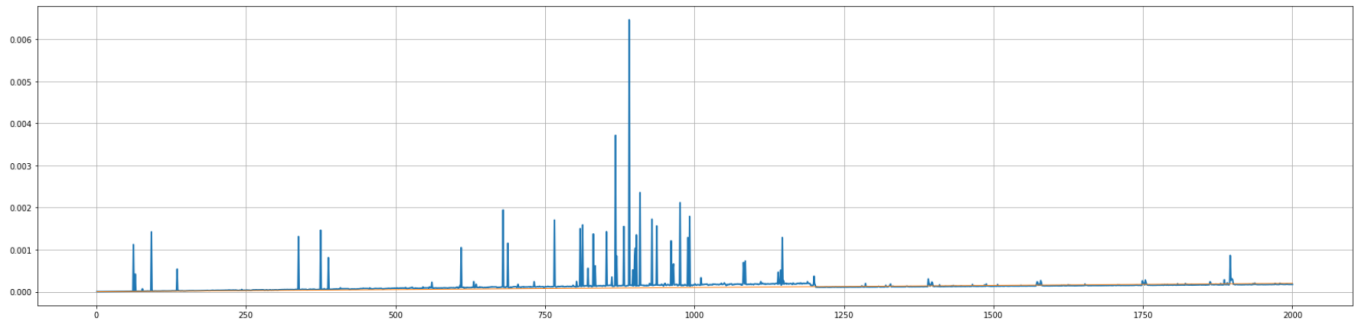
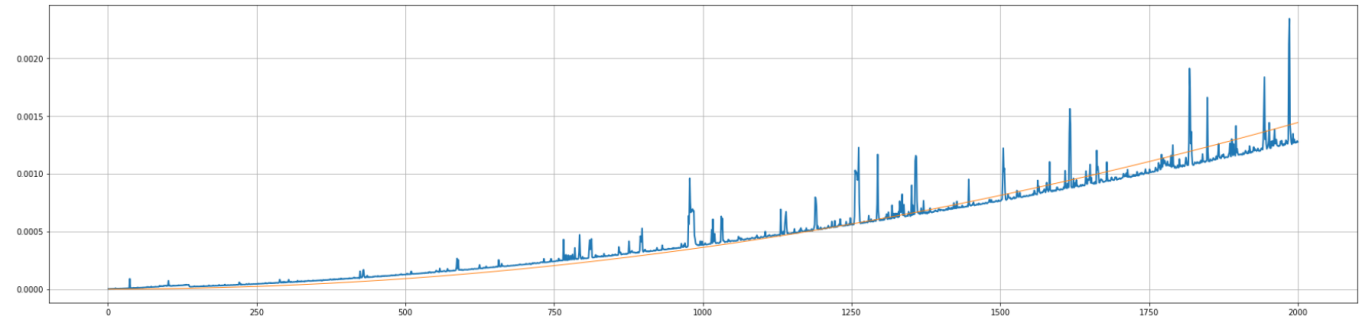


Image 4

2) The sum of elements: $f(v) = \sum_{k=1}^{n} v_k$ with average time complexity *O(n)*. On graph there are a few deviations but average values is close to theoretical ones.

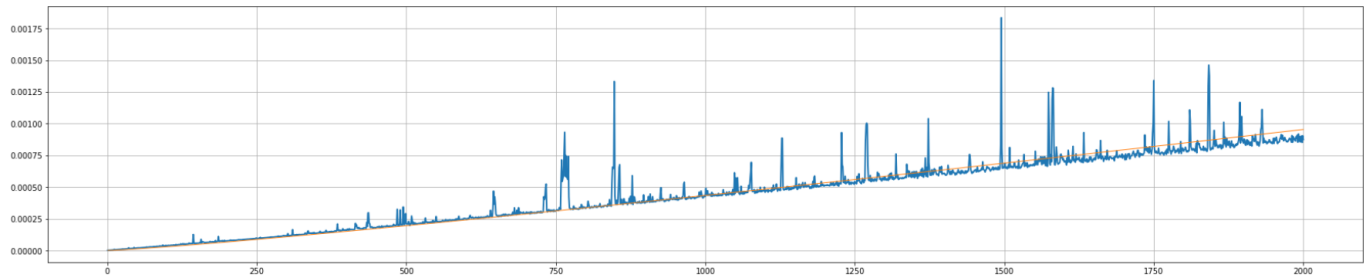


3) The product of elements: $f(v) = \prod_{k=1}^{n} v_k$ with average time complexity $O(n^2)$. As on located above graph values striving to theoretical curve.

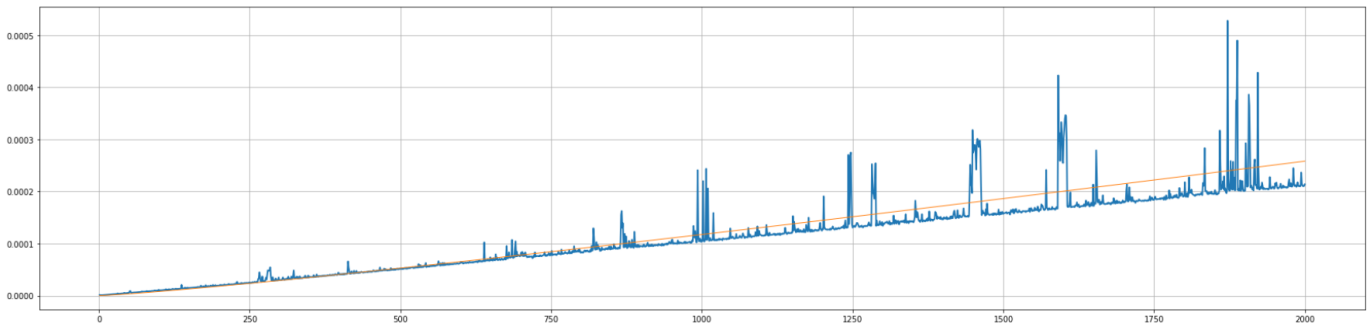


4) Direct calculation of polynomial of degree *n-1*:

$P(x) = \sum_{k=1}^{n} v_k x^{k-1}$, $where\ x = 1.5$ with average time complexity $O(n * log\ n)$.

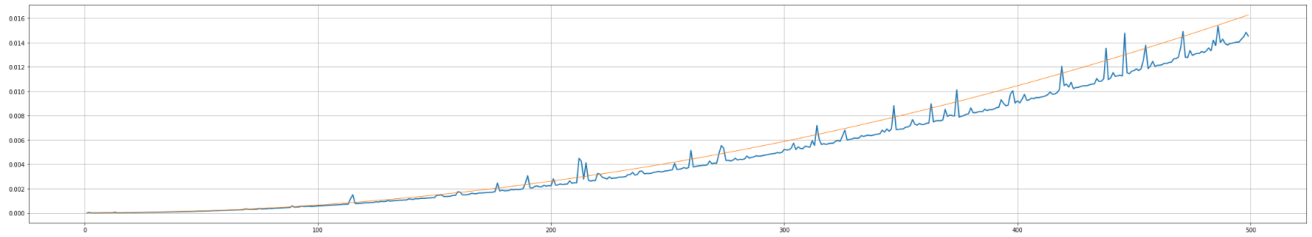


Horner's method: $P(x) = v_1 + x(v_2 + x(v_3 + \ldots))$, $where\ x = 1.5$ with average time complexity $O(n * log\ n)$.

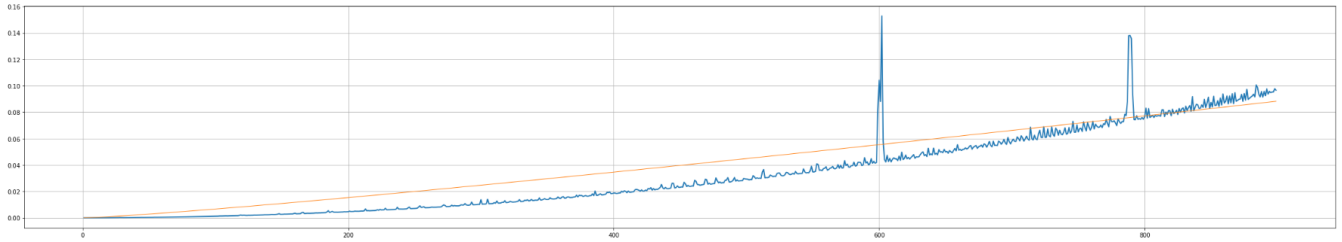


Both of direct and Horner's methods take similar empirical results which close to an theoretical average time complexity $O(n * log\ n)$.

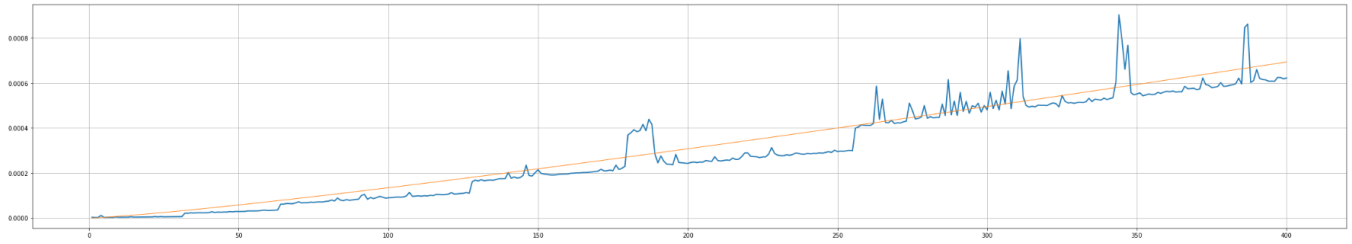
5) Buble sort shows expected results.



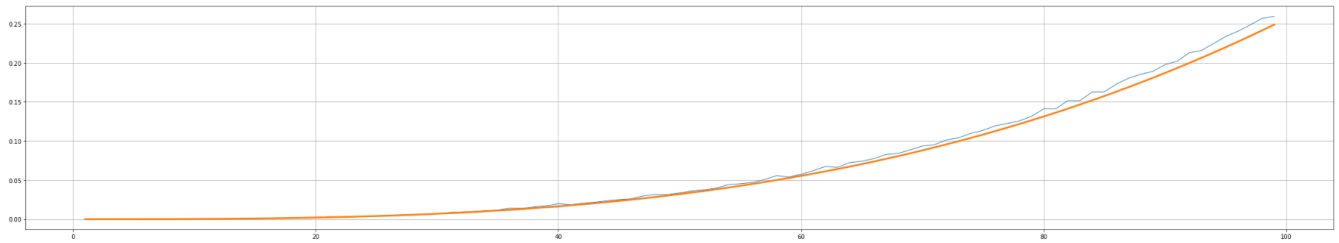
6) Quicksort is one of the best choice for large data sets. But this implementation has limit equal to *n=900*.



7) The results of Timsort algorithms seeks to curve from both sides.



II. The usual matrix product of $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$, $\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$ calls $\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$ where $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$. The product of two matrices will be defined if the number of columns in the first matrix is equal to the number of rows in the second matrix. If the product is defined, the resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.



III. Bubble sort is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. Only list is used in this algorithms as a data structure for storage and replacing elements. The method works by examining each set of adjacent elements in the string, from left to right, switching their positions if they are out of order. The algorithm then repeats this process until it can run through the entire string and find no two elements that need to be swapped. This algorithm design techniques calls brute-force.

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending. To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Conclusions

In conclusion, in this work was considered varied algorithms in terms of average-case time complexity and was visualized the results by matplotlib, Python library. All of experimental values are close to theoretical ones excluding little noize. As a result, during work we repeat and solidify knowledge about space and time complexity, their types and ways to compute.

Appendix

https://github.com/JaneKKTTme/analysis-and-development-of-algorithms