FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report

on the practical task No.1

«Experimental time complexity analysis»

Performed by:
Putnikov Semyon
J4132c

Accepted by:
Dr Petr Chunaev

St.Petersburg
2020

1. Goal

Experimental study of the time complexity of different algorithms.

2. 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.

1. Generate an n-dimensional random vector $\vec{v}$ with non-negative elements. For $\vec{v}$ implement the following calculations and algorithms:

   (a) $f(v) = const$ *(constant function)*

   (b) $f(v) = \sum_{k=1}^{n} v_k$ *(the sum of elements)*

   (c) $f(v) = \prod_{k=1}^{n} v_k$ *(the product of elements)*

   (d) supposing that the elements of $\vec{v}$ 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 $P(x) = v_1 + x(v_2 + x(v_3 + ...))$;

   (e) Bubble Sort of the elements of $\vec{v}$;

   (f) Quick Sort of the elements of $\vec{v}$;

   (g) Timsort of the elements of $\vec{v}$;

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

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

3. Brief theoretical part

3.1 Horner's method

Horner's method is a method for approximating the roots of polynomials that was described by William George Horner in 1819.

Given the polynomial $p(x) = \sum_{k=1}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_n x^n$ where $a_0, ...a_n$ are constant coefficients, the problem is to evaluate the polynomial at a specific value x_0 of x.

For this, a new sequence of constants is defined recursively as follows:

$$\begin{aligned} b_n &:= a_n \\ b_{n-1} &:= a_{n-1} + b_n x_0 \\ &\vdots \\ b_1 &:= a_1 + b_2 x_0 \\ b_0 &:= a_0 + b_1 x_0 \end{aligned}$$

Then b_0 is the value of $p(x_0)$. To see why this works, the polynomial can be written in the form

$$p(x) = a_0 + x\Big(a_1 + x\Big(a_2 + x\Big(a_3 + ... + x\big(a_{n-1} + xa_n\big)...\Big)\Big)\Big)$$

3.2 Bubble sort

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. The algorithm is named for the way smaller or larger elements "bubble" to the top of the list. Best-case perfomance is $O(n)$ and worst-case is $O(n^2)$.

3.3 Quick sort

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two

sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting. Best-case perfomance is $O(n)$ (three-way partition and equal keys) and worst-case is $O(n^2)$ (one of the sublists returned by the partitioning routine is of size $n - 1$ and pivot happens to be the smallest or largest element in the list).

3.4 Timsort

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. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled. Best-case perfomance is $O(n)$ and worst-case is $O(n \log n)$.

4. Results

4.1 Constant function (Task I, subtask a)

At Figure 1 we can see execution time of constant function. This plot tends to direct line as expected, but have some outliers. It is a result of cpu overheating which affects execution process.

Nevertheless, the graph proofs that the execution time of constant function does not depend on the number of variables.

4.2 The sum of elements (Task I, subtask b)

At Figure 2 we can see that execution time is similar to constant function. Some outliers, caused by cpu overheating also included, but the main thing is that plot tends to direct line. This happens because of using embedded Python function for elements sum.

So on practice we have something close to constant complexity, not expected $O(n)$

4.3 The product of elements (Task I, subtask c)

At Figure 3 we can see that execution time is also similar to constant function. We have same situation, as in 4.3. We also use embedded Python function, but for elements product this time. On practice we have time complexity, which is close to constant, instead of expected $O(n)$.

4.4 Polynomial calculation (Task I, subtask d)

Direct way. At Figure 4 we can see that time complexity is $O(n)$. Theoretically it should be $O(n^2)$, but implementation has no simple loop for evaluation of x, it uses embedded programming language function.

Horner's method. At Figure 5 plot shows us that time complexity is $O(n)$. This is just as we expected theoretically. Methods includes result initialization as coefficient of x^n, repeatedly result with x multiplication and adding next coefficient to result. Finally we return result.

So we have $O(n)$ both theoretically and on practice.

4.5 Bubble Sort (Task I, subtask e)

At Figure 6 we see, that time on plot increases closely to $O(n^2)$. This complexity is theoretically expected, because we have to make $O(n^2)$ comparisons.

4.6 Quick Sort (Task I, subtask f)

At Figure 7 we see, that time on plot increases closely to $O(n \log n)$. This time complexity is average theoretical perfomance, so our practical result completely matches it (excluding cpu-caused outliers).

4.7 Tim Sort (Task I, subtask g)

At Figure 8 we see, that time on plot, as in previous case, increases closely to $O(n \log n)$. This time complexity is also known as average theoretical for this algorithm, so our practical result matches it.

4.8 Matrices product (Task II)

Despite all cpu outliers the plot at Figure 9 clearly shows us, that time complexity for matrices multiplication is $O(n^3)$. This time complexity matches theoretical one for this algorithm, so our practical result proves it.

4.9 Data structures and design techniques (Task III)

We use embedded Python data structures at programming realization of above described algorithms, such as dictionaries and lists. Also it was very helpful to use Dataframe structure from «pandas» library.

For measuring execution time we use method from library «timeit». We use library «matplotlib» to create plots.

As design technique we use pattern decorator and recursive architecture.

5. Conclusions

To sum it up, we've measured the average execution time of well-known algorithm implementations, using data plots, representing our time values as a function of n.

After comparison of the empirical and theoretical time complexities we can say, that sometimes features of programming language helps us to speed up calculations. But also sometimes hardware makes it's adjustments, so we don't have stable time complexity on practice, only approximated values.

6. Figures

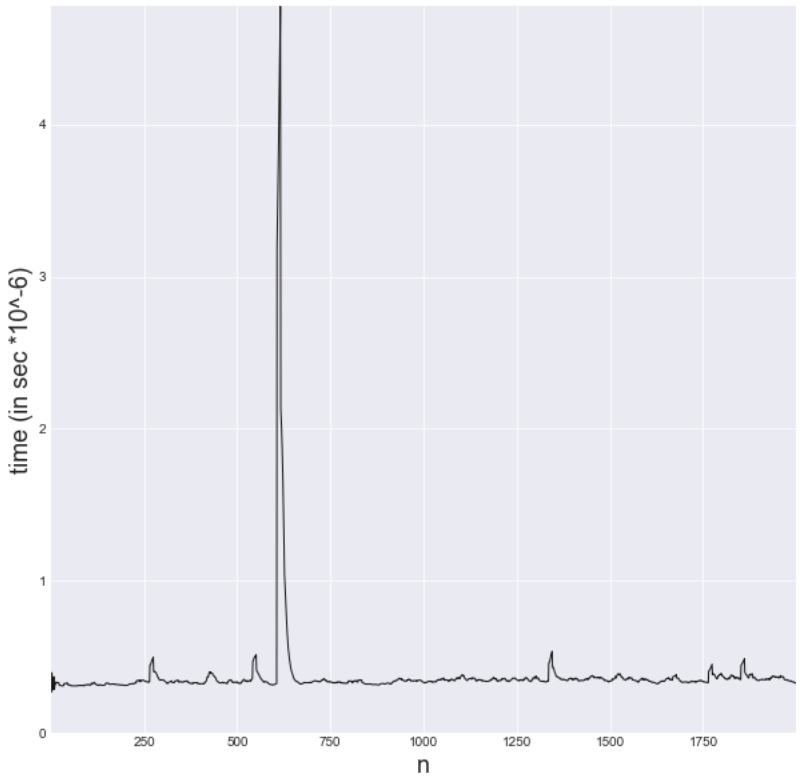

Figure 1: Plot of constant function.

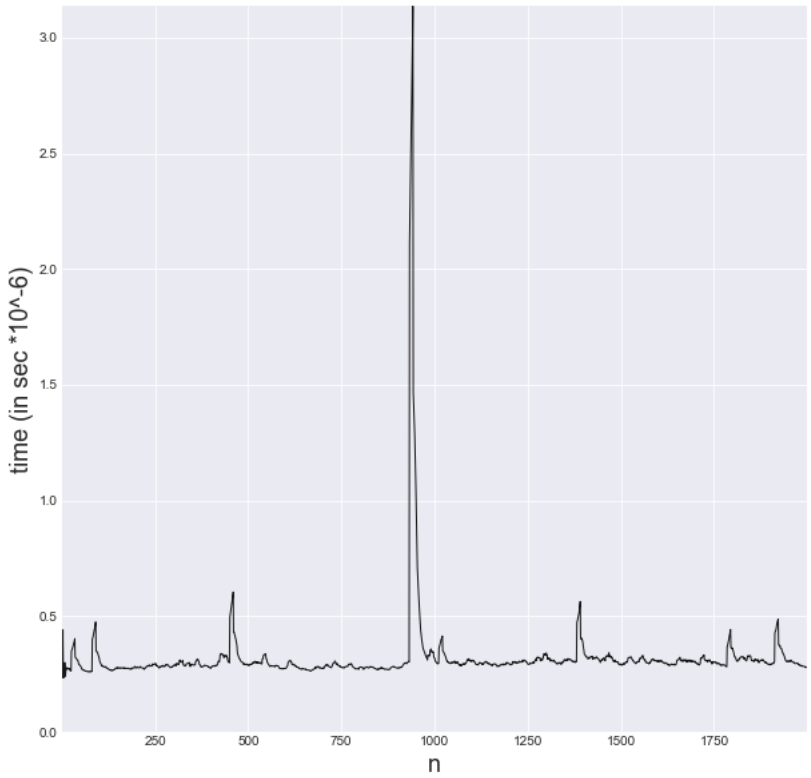

Figure 2: Plot for sum of elements.

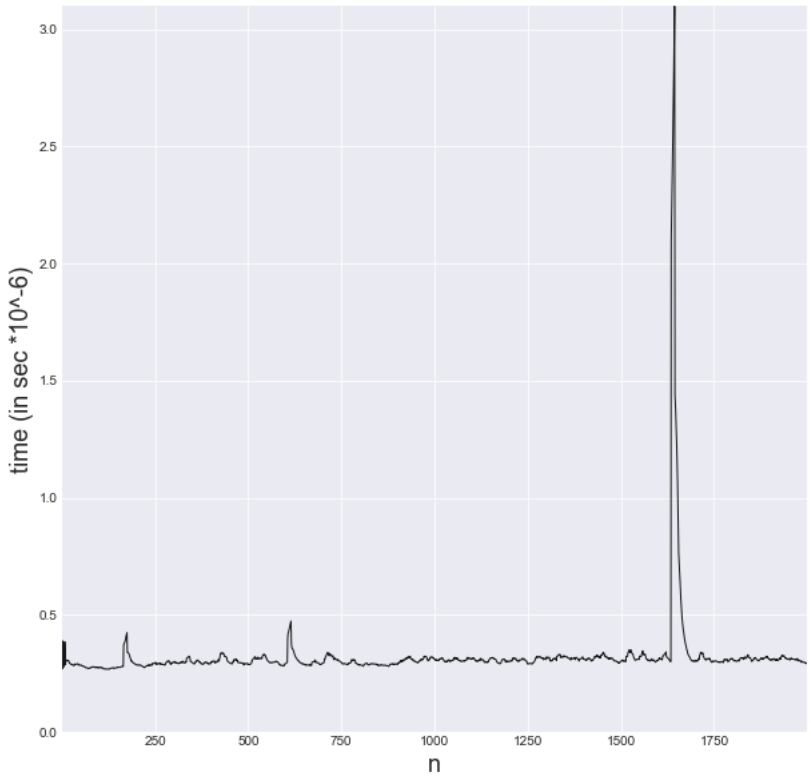

Figure 3: Plot for product of elements.

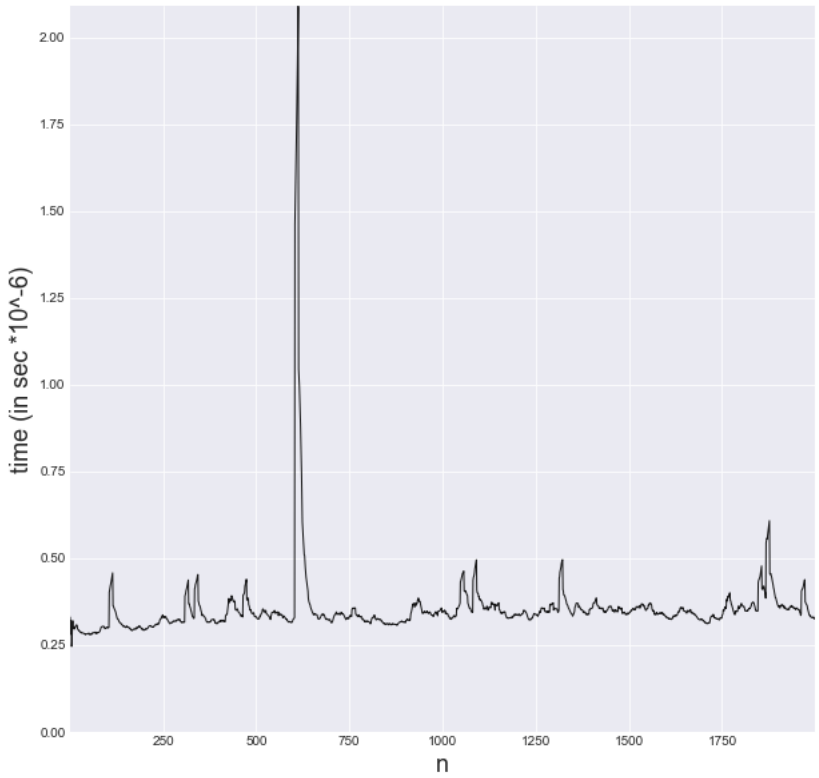

Figure 4: Plot for direct polynomial calculation.

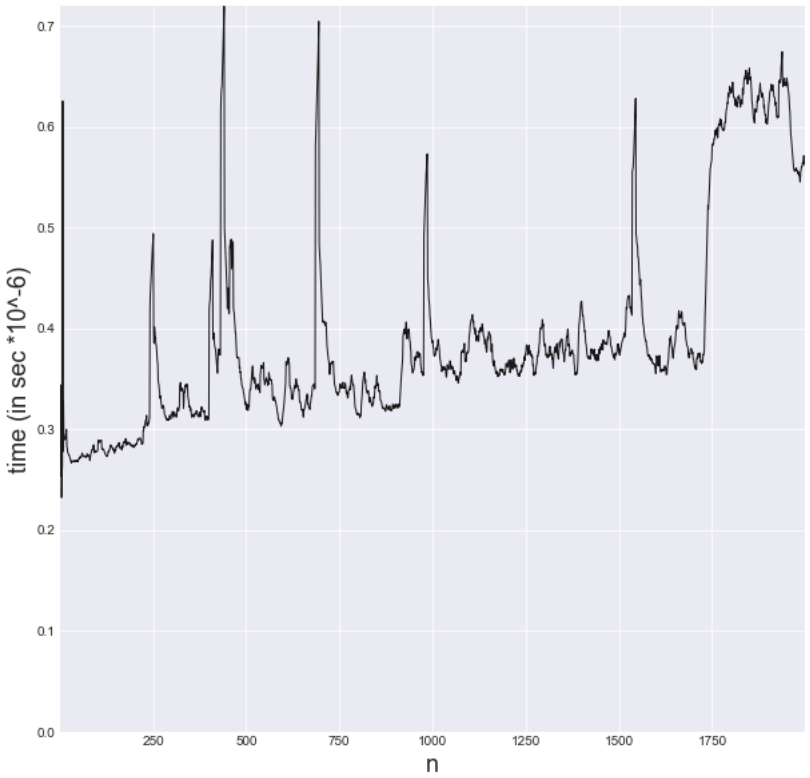

Figure 5: Plot for polynomial calculation by Horner's method.

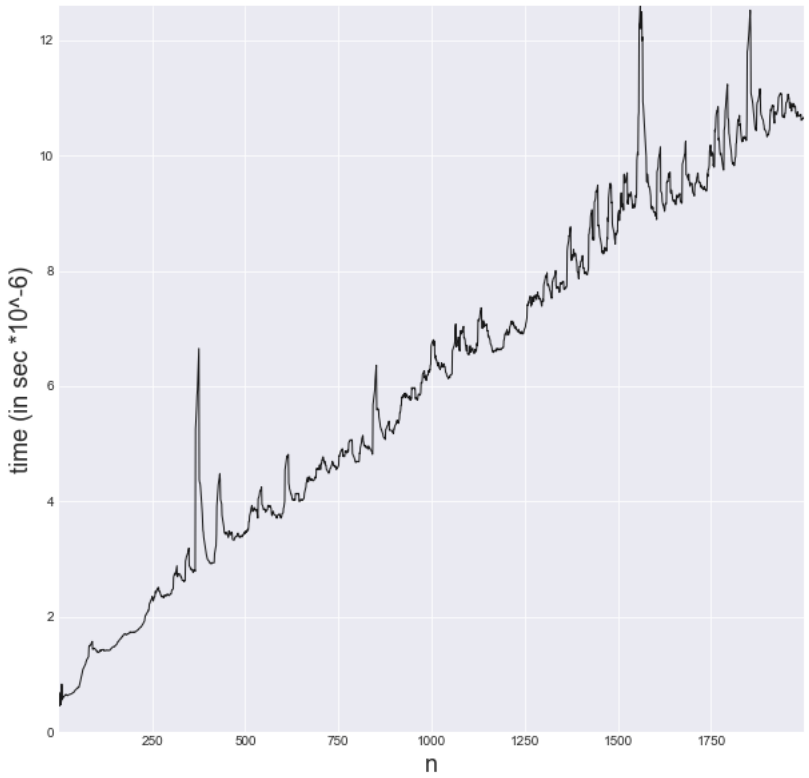

Figure 6: Plot for "bubble" sorting.

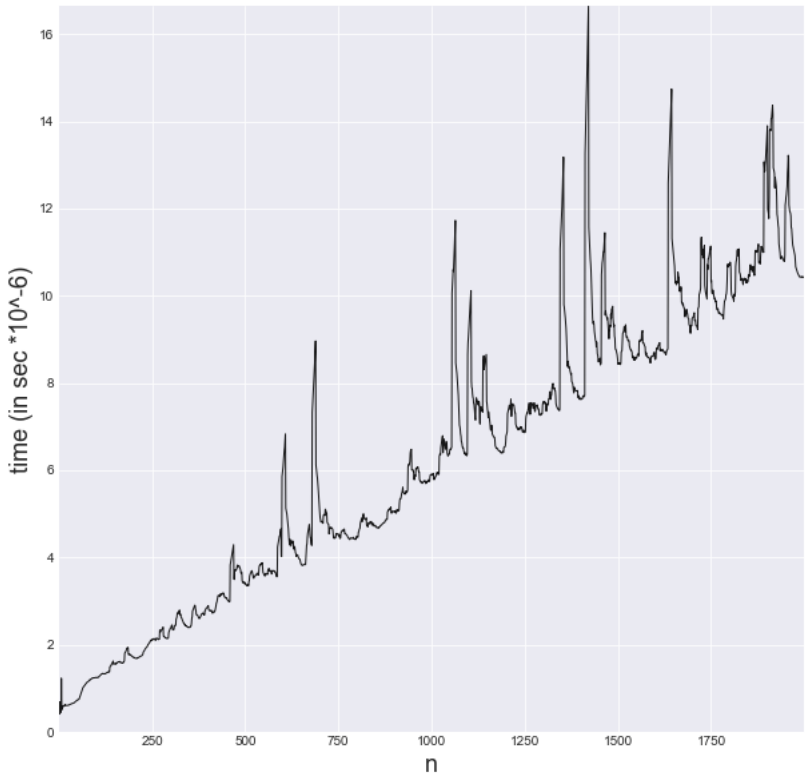

Figure 7: Plot for "quick" sorting.

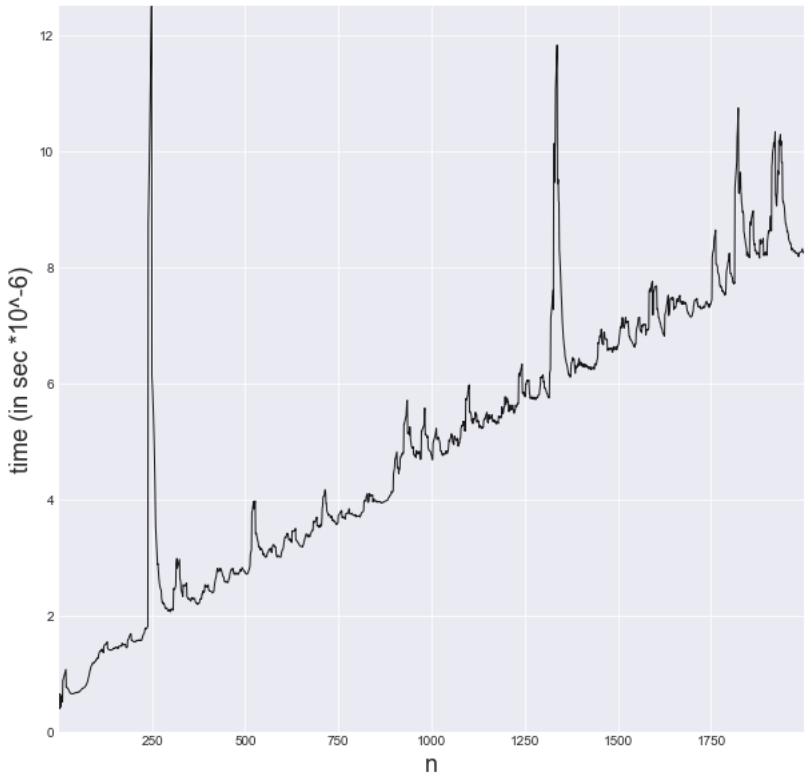

Figure 8: Plot for "Tim" sorting.

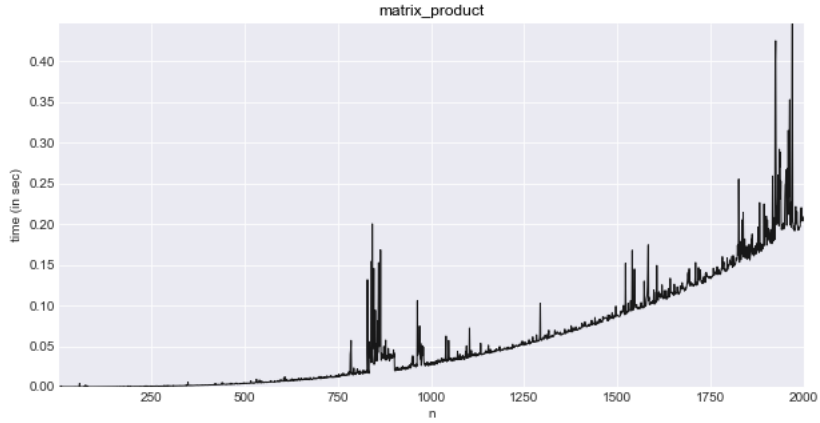

Figure 9: Plot for product of two random matrix.