FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

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

Performed by
Anastasia Miroshnikova
j4133c
Accepted by
Dr Petr Chunaev

St. Petersburg
2020

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 $v=[v_1, v_2, \ldots, v_n]$ with non-negative elements. For v, implement the following calculations and algorithms:

1) $f(v)=const$ (constant function);
2) $f(v)=\sum_{k=1}^{n} v_k$ (the sum of elements);
3) $f(v)=\prod_{k=1}^{n} v_k$ (the product of elements);
4) supposing that the elements of 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 v;
6) Quicksort of the elements of v;
7) Timsort of the elements of 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

Constant function is a function whose (output) value is the same for every input value. The time complexity of constant function is O(1).

Functions **Sum(Array)** and **Prod(Array)** require looking at each element once, so time complexity of those functions is O(n).

Calculation of the **value of polynomial** P of degree $n-1$ with coefficients $v=[v_1, v_2, \ldots, v_n]$ can be done in several ways. One of them is **direct** – calculate values of array $x_n=[x^0, x^1, x^2, \ldots, x^{n-1}]$ and multiply them by coefficients v. We can subsequently calculate values of x_n and keep the last value to avoid repeating multiplications, so this method will require only O(1) additional memory and has O(n) time complexity. The second method we have to try is **Horner's recursive method** – at the first step we take the last coefficient (multiplier of x^{n-1}), at the *i*-th steps we multiply the previous value by x and add the multiplier of x^{n-i-1}. This method also requires O(1) additional memory and has O(n) time complexity.

Bubble sort, sometimes referred to as **sinking 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 has two loops – outer that runs through elements *from 1 to n*, and inner, that runs through elements *from i to n*, where *i* is the element number from outer loop. Each operation within the inner loop has time complexity O(1), that is, the overall time complexity is the sum of all operations in the outer and inner loop, and can be derived as $\sum_{i=1}^{n}\sum_{j=i}^{n} j=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}=\frac{n^2}{2}+\frac{n}{2}$, i.e. $O(n^2)$.

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. Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is very rare. The disadvantage of the algorithm is that it is unstable – it doesn't preserve the order of elements in the initial array.

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 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. In the worst case, Timsort takes O(n log n) comparisons to sort an array of n elements. In the best case, which occurs when the input is already sorted, it runs in O(n) time. It is advantageous over Quicksort for sorting object references or

pointers because these require expensive memory indirection to access data and perform comparisons and Quicksort's cache coherence benefits are greatly reduced.

Matrix multiplication is a multiplication of two matrices – A with shape (k, l), and B with shape (l, m), producing matrix C with shape (k, m) such that

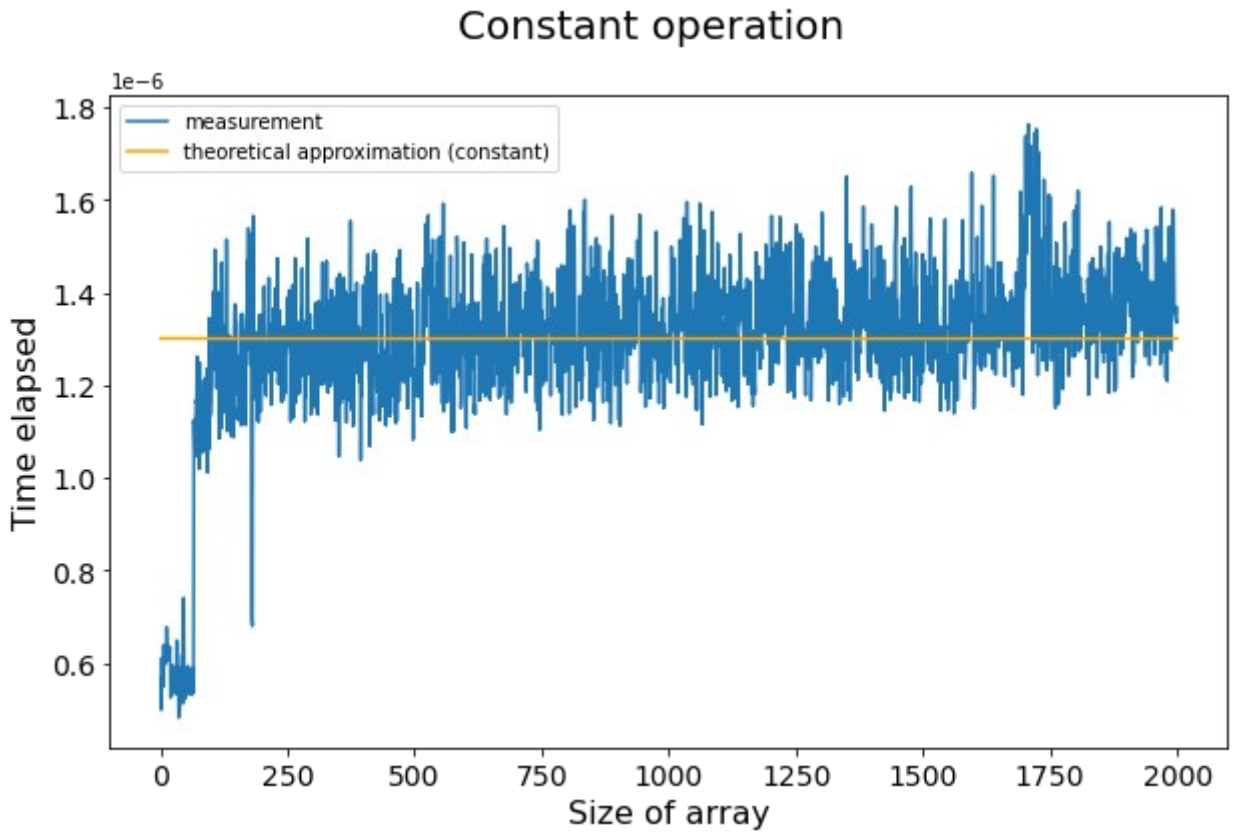
$c_{ij}=a_{i1}b_{1j}+a_{i2}b_{2j}+...+a_{in}b_{nj}=\sum_{k=1}^{n} a_{ik}b_{kj}$. The time complexity is O(k l m), or O(n^3) in case of square matrices with shape (n, n).

Results

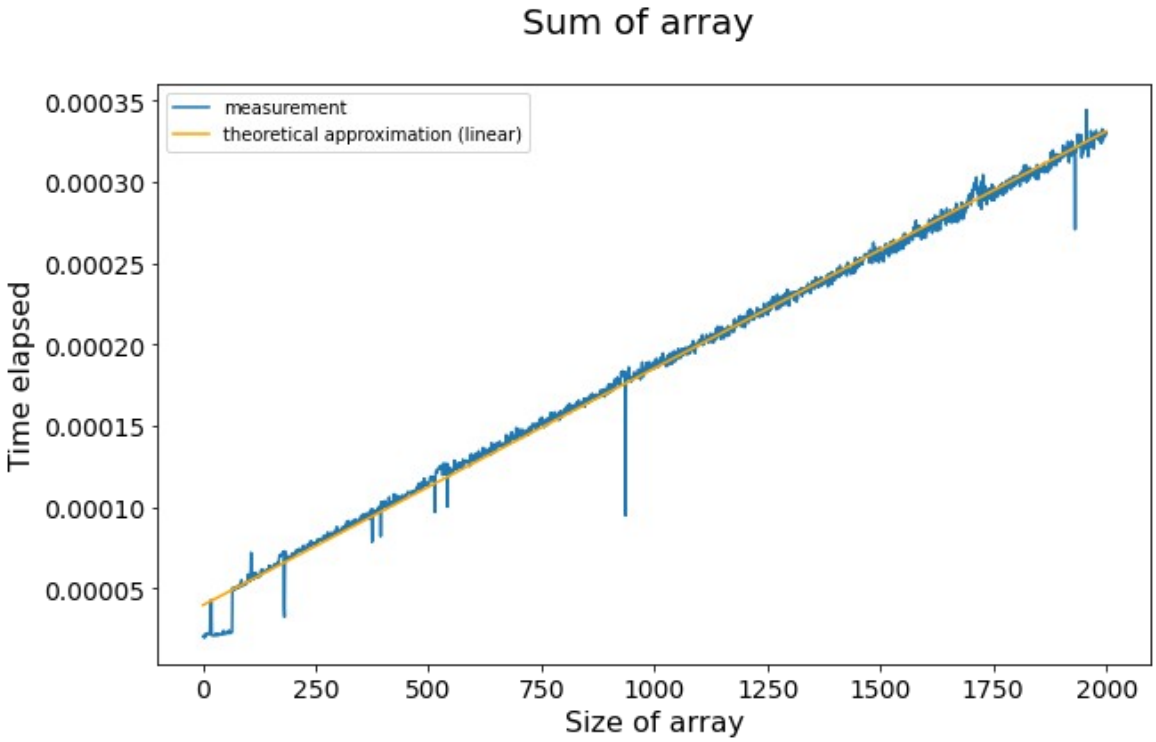
The results were obtained using Python 3.6.9 programming language. The device used was a notebook with fixed CPU frequency 1.2 GHz.

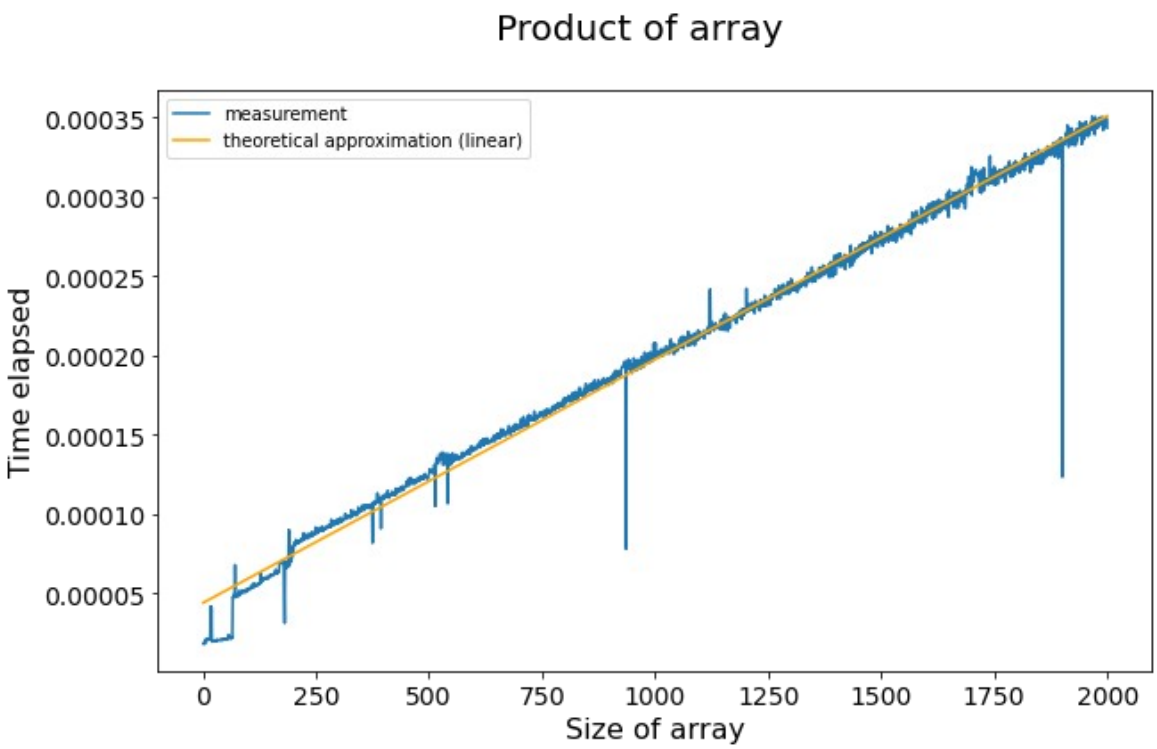
For each value of *n* in range from 1 to 2000 7 arrays or matrices were generated, then each function was called on the array once. The time elapsed was measured using built-in *timeit* Python library. Then maximal and minimal times were deleted and the mean time was collected.

Theoretical approximations were obtained by using *numpy.polyfit* function from *numpy* Python library. The degree on the polynomial was set to 1 (except for constant function, where it was set to 0), and the *x* array was preprocessed in accordance with theoretical expectation – for Bubble sort I've taken x^2 as predictor, for Quicksort and Timsort - $x \cdot \ln(x)$, for matrix multiplication – x^3.

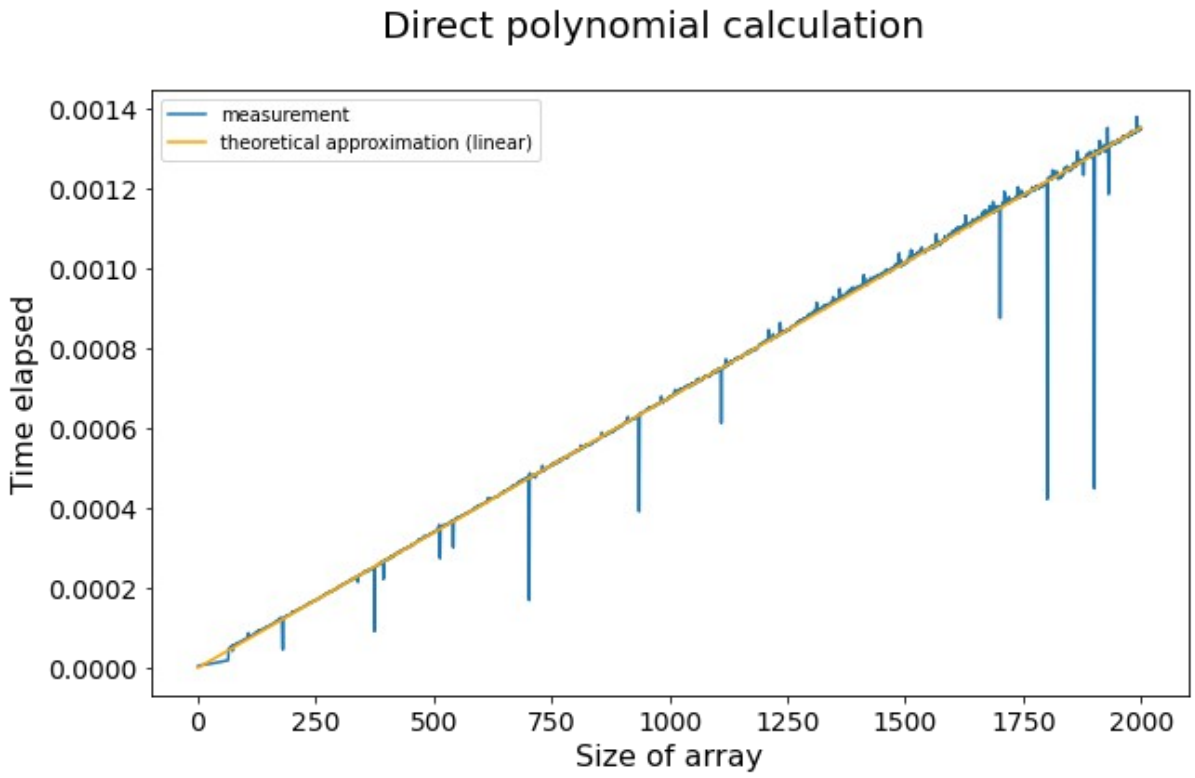


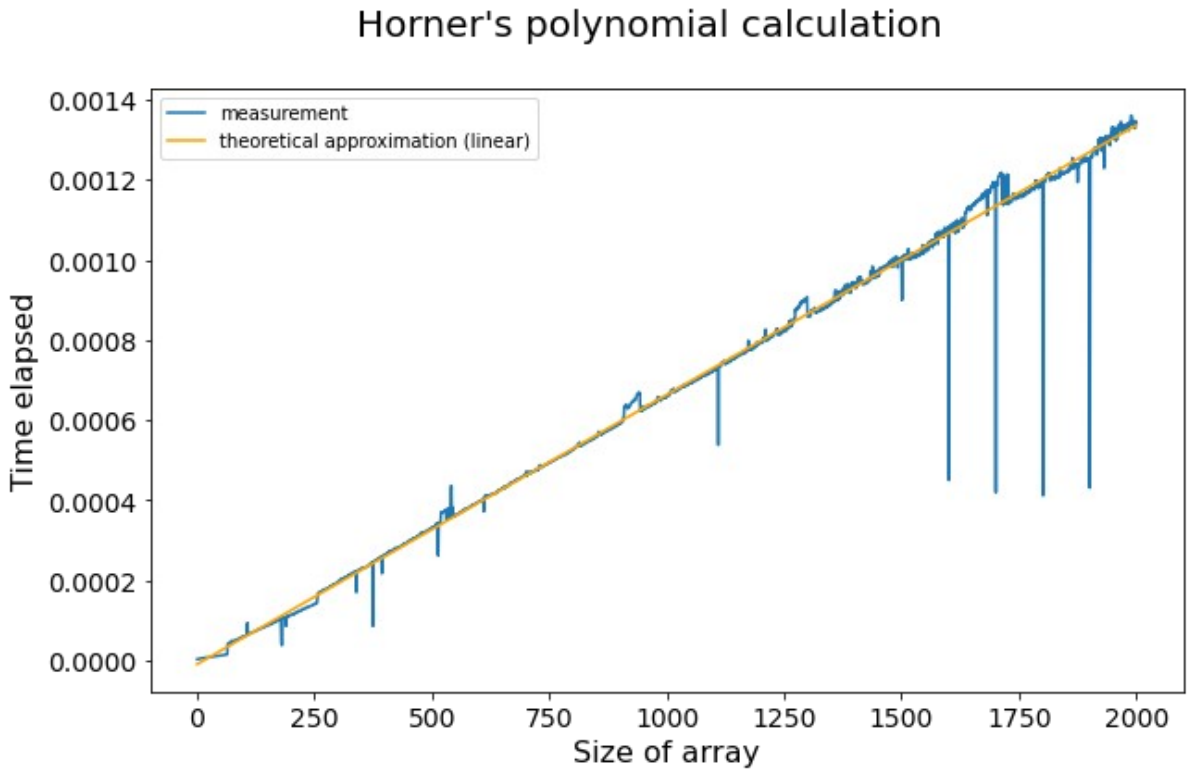
Time required for constant function didn’t depend on size of an array and required about 1.36 µs.



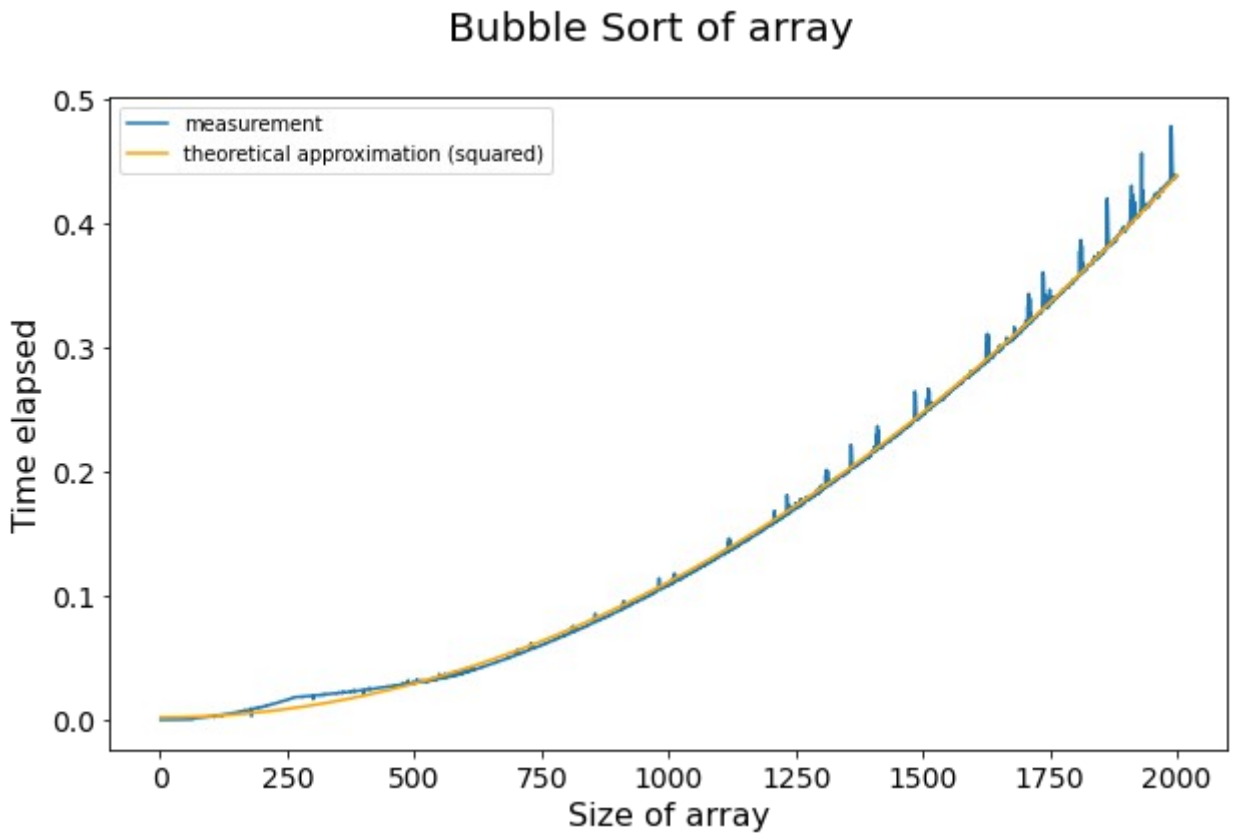


The functions sum of array and product of array had linear time complexity as expected.

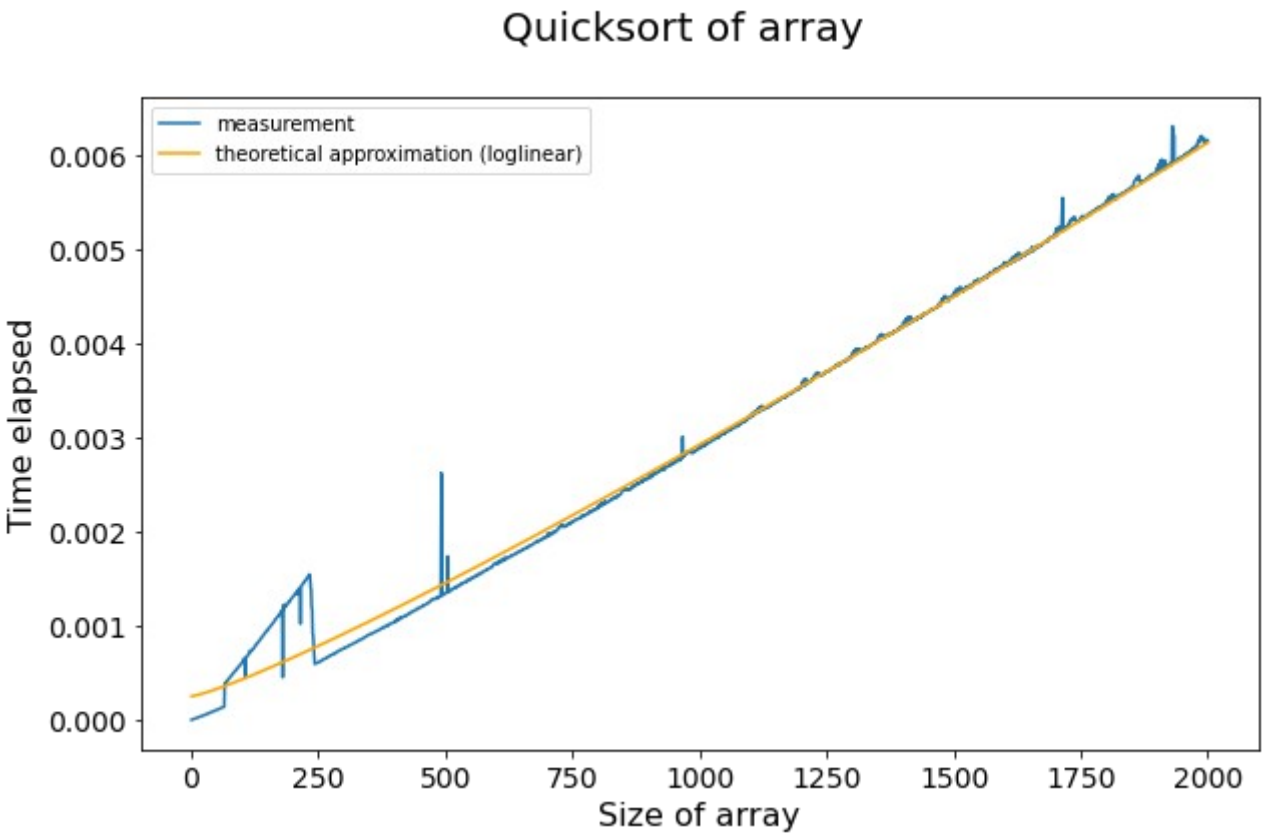


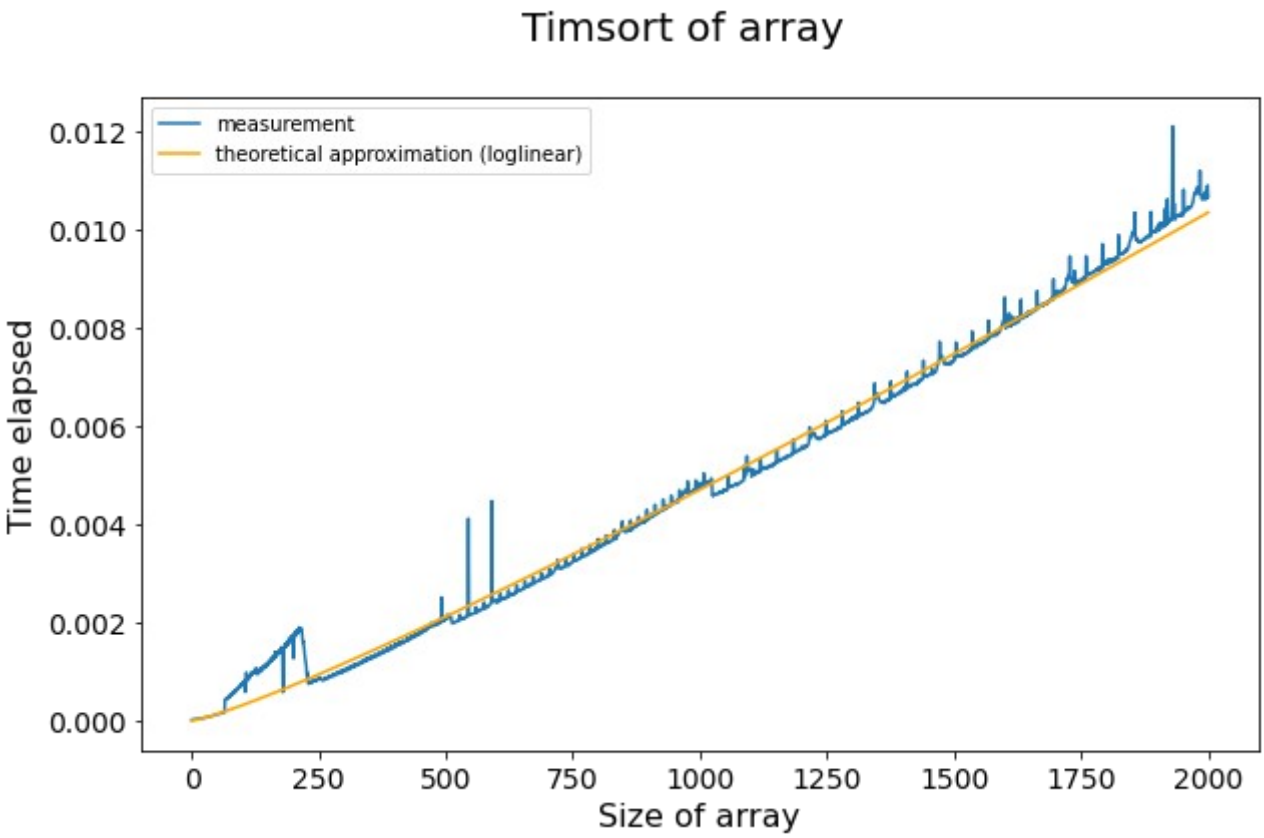


Both direct and Horner's methods for polynomial function calculation shows linear time complexity.

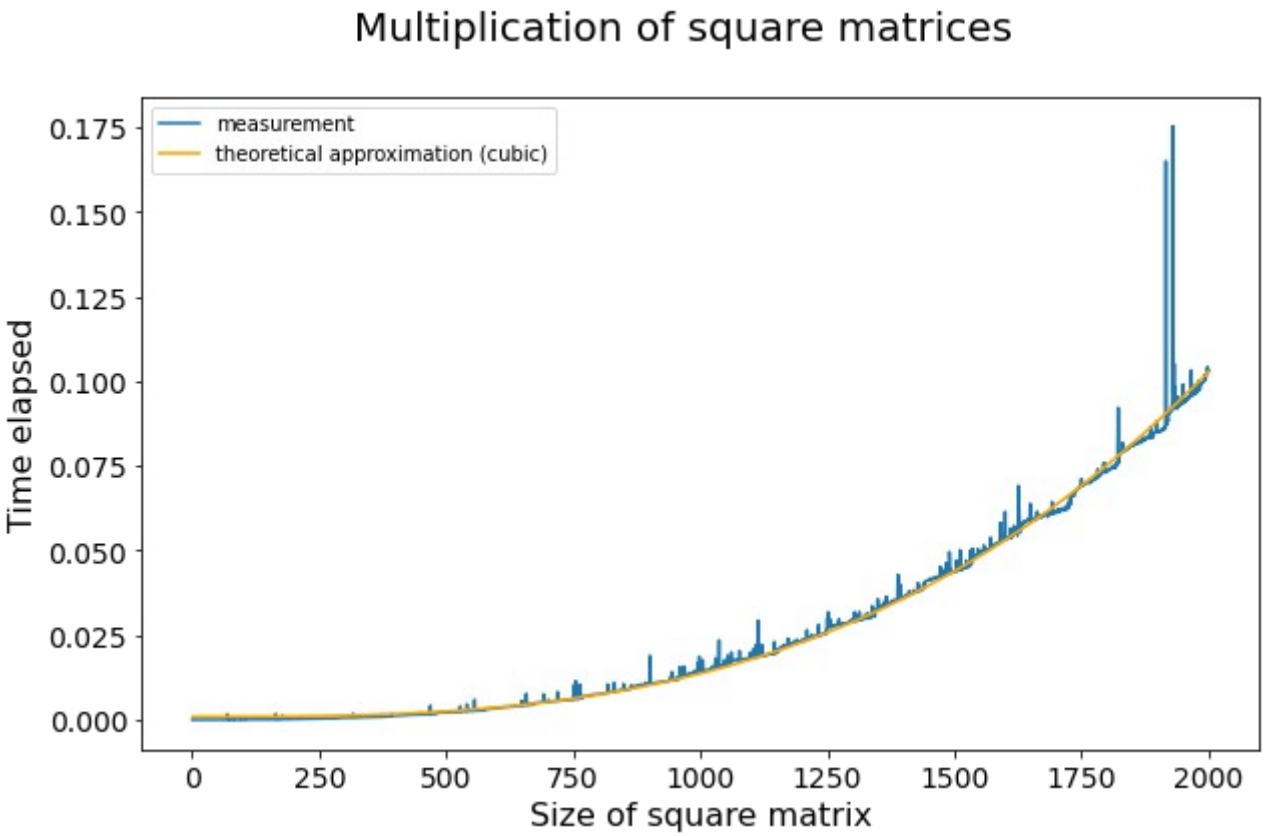


Bubble sort perfectly fits its theoretical squared approximation.





Both Quicksort and Timsort in theory should have loglinear time complexity, but the selected functions do not fit it well. One of the reasons is the initial range of sizes (from 1 to 250) where the linear coefficient was visually higher than on the whole graph. I don't really know what happened there. I manually set the frequency of the CPU, so it shouldn't have changed much during the experiment.



Multiplication of square matrices takes cubic time with respect to matrix size.

Conclusions

The goal of this work was to measure time required for several known algorithms and compare it to the known theoretical time complexity. Analysis of the results have shown that they are in concordance with the expected values.

Appendix

Source code can be found at https://github.com/Miffka/algo_itmo/tree/master/task1.