FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

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

Performed by
Tiulkov Nikita
J4133c

Accepted by
Dr Petr Chunaev

St.Petersburg
2021

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 v with non-negative elements. For 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 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 v;
   (f) Quick Sort of the elements of v;
   (g) Timsort of the elements of 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

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:

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

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

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

3.2. Bubble sort

Bubble sort is a simple and common 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 $O(n)$ (when our array is already sorted) and worst-case is $O(n^2)$ (when we need to replace almost all elements).

3.3. Quicksort

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), on average, the algorithm takes $O(n \log n)$ comparisons to sort n items, 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. I, Constant function

At Figure 1 we can see execution time of constant function. This plot tends to direct line as expected, but have some ”outliers”. These ”outliers” are caused by computer limitations.

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

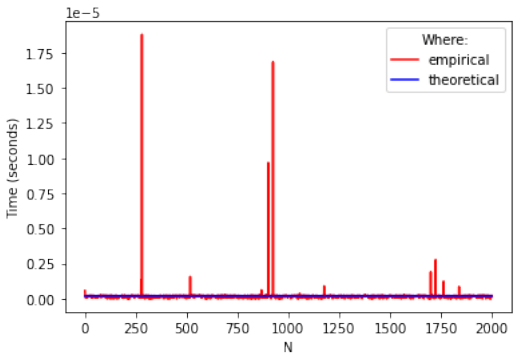

Figure 1: Constant function plot

4.2. I, The sum of elements

At Figure 2 we can see that execution time of sum of elements. Some outliers, caused by computer limitations, but plot tends to linear function as expected. And we have a $O(n)$ time complexity.

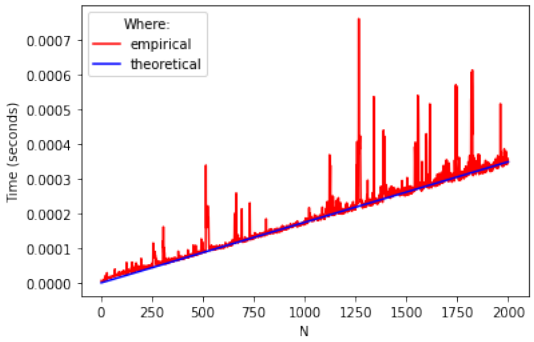

Figure 2: The sum of elements plot

4.3. I, The product of elements

At Figure 3 we can see that execution time is also similar to sum of elements. As expected we have tends to linear function and $O(n)$ time complexity.

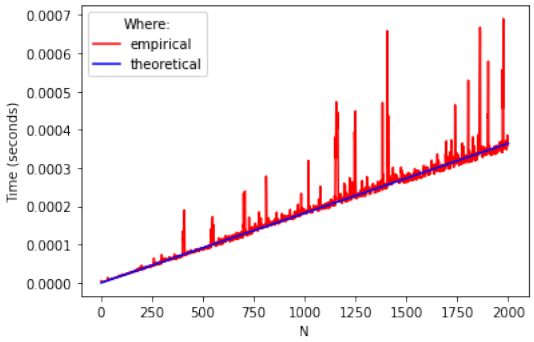

Figure 3: The product of elements plot

4.4. I, Polynomial calculation

Naive method. At Figure 4 we can see that time complexity is $O(n^2)$. Via realization with two loops - first with results calculation and second with product of x's. It is confirms of theoretical estimation.

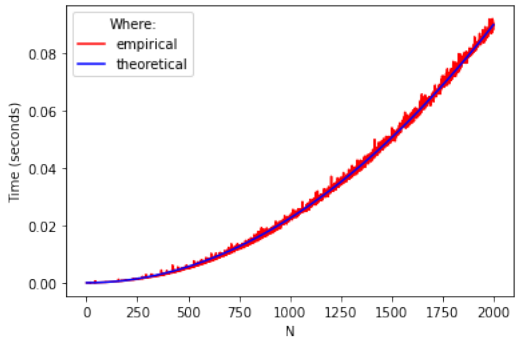

Figure 4: Naive polynomial calculation plot

Horner's method. At Figure 5 plot shows us that time complexity is $O(n)$. This is just as we expected theoretically. Compare with naive method we have only one loop for execution of result.

So we have $O(n)$ both theoretically and on practice (except of "outliers").

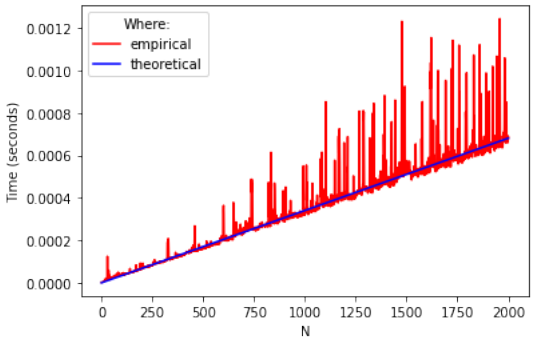

Figure 5: Horner's method plot

4.5. I, Bubble Sort

At Figure 6 we see, that time on plot increases closely to $O(n^2)$. So we have $O(n^2)$ both theoretically and on practice.

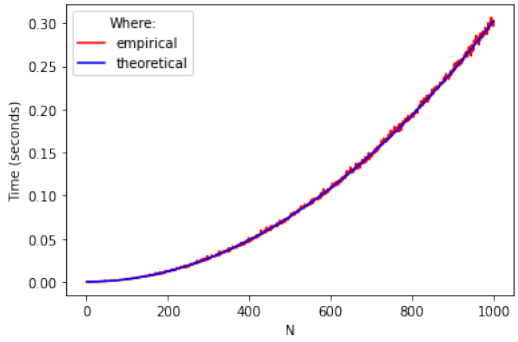

Figure 6: Bubble sort plot

4.6. I, Quicksort

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 matches with it (except of "outliers").

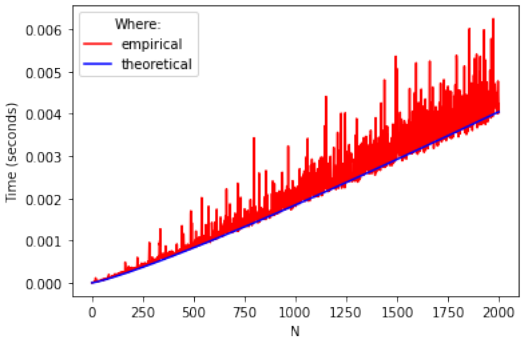

Figure 7: Quicksort plot

4.7. I, Timsort

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.

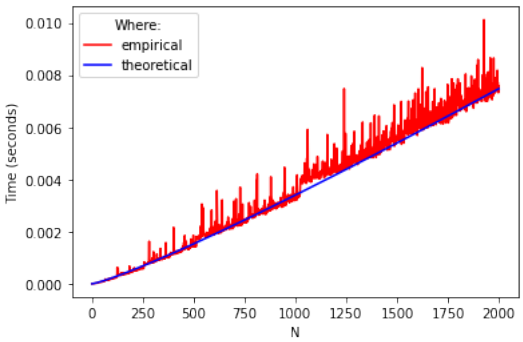

Figure 8: Timsort plot

4.8. II, Matrices product

At Figure 9 we see that experimental and theoretical results almost match. So we have $O(n^3)$ time complexity for both cases. In that case we use not optimized multiplication but naive method.

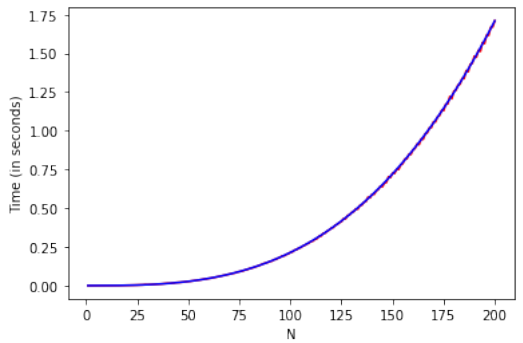

Figure 9: The product of two random matrix plot

4.9. III, Data structures and design techniques

In this task we have used data structures like list (one dimension - vector and multiple dimension - matrix). This list is provided via Python language.

As design technique we have used brute-force technique for bubble sort algorithm, divide and conquer technique for quicksort algorithm and for timsort algorithm - decrease and conquer (insertion sort) + divide and conquer (merge sort).

5. Conclusions

In this task, the time complexity of various algorithms was investigated with Python language support via using of embedded data structures and plotting methods.

It can be noted that the trend of empirical estimation of time complexity coincides with the theoretical one. However, ”outliers” are visible in empirical graphs due to computer implementation.

Appendix

The code of algorithms you can find on GitHub: