FEDERAL STATE AUTONOMOUS EDUCATIONAL
INSTITUTION OF HIGHER EDUCATION ITMO UNIVERSITY

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

Performed by

Rami Al-Naim

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

In order to obtain data to study the time complexity of different algorithms the following was done. For each n from 1 to 2000, the average computer execution time was measure for of programs implementing the algorithms and functions below for five runs. The experimental elapsed time data was plotted in respect to n. Then the resulted experimental data was approximated with curves based on the assumptions of algorithm's time complexity.

The more formal description of the algorithm: I. Generate an n-dimensional random vector $v = [v_1, v_2, ..., v_n]$ 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 $P(x) = \sum_{k=1}^{n} v_k x^{k1}$ (i.e. evaluating each term one by one) and by Horner's method by representing the polynomial as $P(x) = v1 + x(v2 + x(v3+))$;
5) Bubble Sort of the elements of v;
6) Quick Sort of the elements of v;
7) Timsort of the elements of v.
II. Generate random matrices A and B of size n 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.

3 Brief theoretical part

Time complexity of an algorithm frequently described using Big O notation. Since exact execution time may vary due to different hardware and other conditions, Big O notation describes the relation between execution time and size of an input data.

For sum of elements, product of elements, polynomial computation using direct computation and Horner's method has $O(n)$ time complexity. Constant function has constant time complexity, or $O(1)$.

Bubble sort is a sorting algorithm with the worst-case time complexity of $O(n^2)$ and $O(n)$ complexity if input data is sorted. The Python implementation of Bubble sort can be seen below.

```
def bubble_sort(v):
    n = len(v)
    for i in range(1, n):
        swapped = False
        for j in range(n - i):
            if v[j] < v[j+1]:
                v[j], v[j+1] = v[j+1], v[j]
                swapped = True
        if not swapped:
            break

    return v
```

The Quicksort is the divide-and-conquer algorithm with time complexity of $O(n^2)$ for worst-case and $nlog(n)$ on average. The idea of the algorithm is to divide the input array in buckets and recursively sort them. The pseudocode for this algorithm is can be seen below.

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

In this work a `numpy`'s implementation of quicksort(`numpy.sort()`).

The Timsort is an algorithm designed for Python language with time complexity of $O(n)$ for best case and $nlog(n)$ for worst case and on average. Timsort uses stack in order to store 'runs' - sequences of ordered data inside input array. During this work a Python implementation of Timsort was used.

Matrix multiplication is a costly operation, and naive approach has complexity of $O(n^3)$ since for getting each value of the resulting matrix each column of the second matrix should be multiplied by row of the first matrix. There are some methods which are less time-costly (for example, Coppersmith-Winograd method). Moreover, time complexity of a matrix multiplication also may depends on the type of matrix: sparse, triangle, diagonal. During this work a `numpy`'s implementation of matrix product was used.

4 Results

In this section the experimental data is plotted and approximated with different curves depending on the algorithm. For all computation and data processing `Python3.6` were used with help of `scipy`, `numpy` and `pandas` modules.

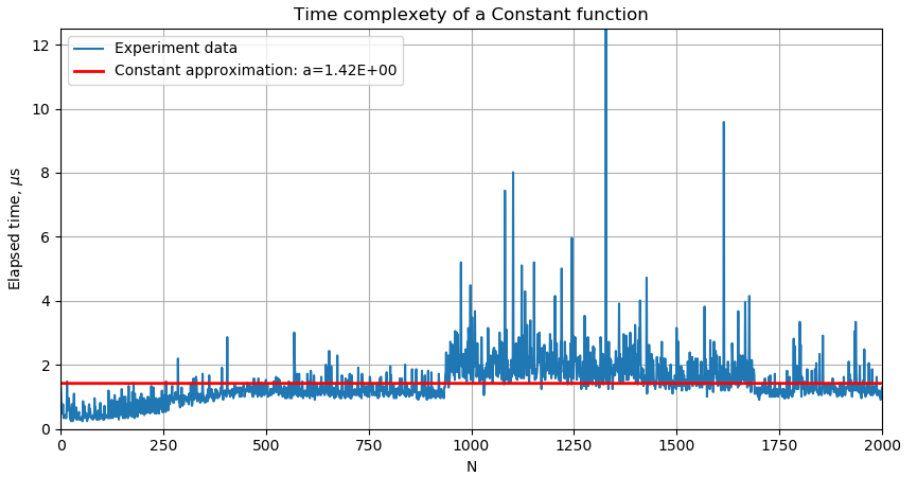

Figure 1: Approximation of experimental data of a constant function

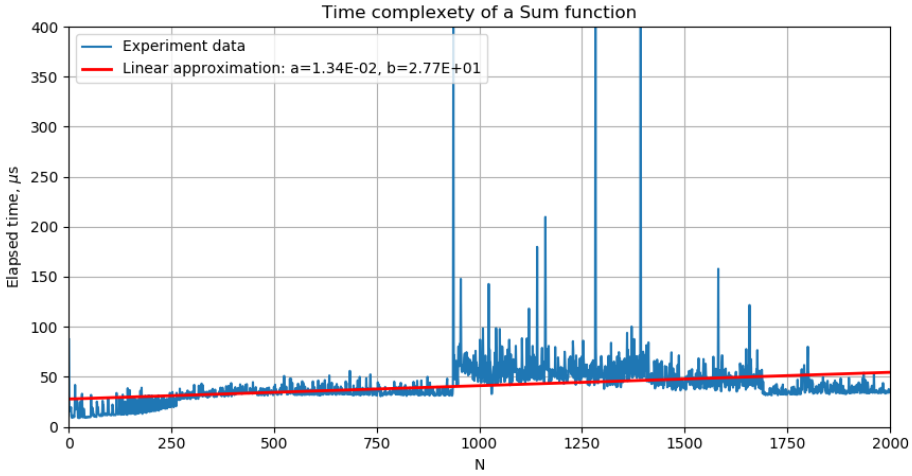

Figure 2: Approximation of experimental data of a summation function

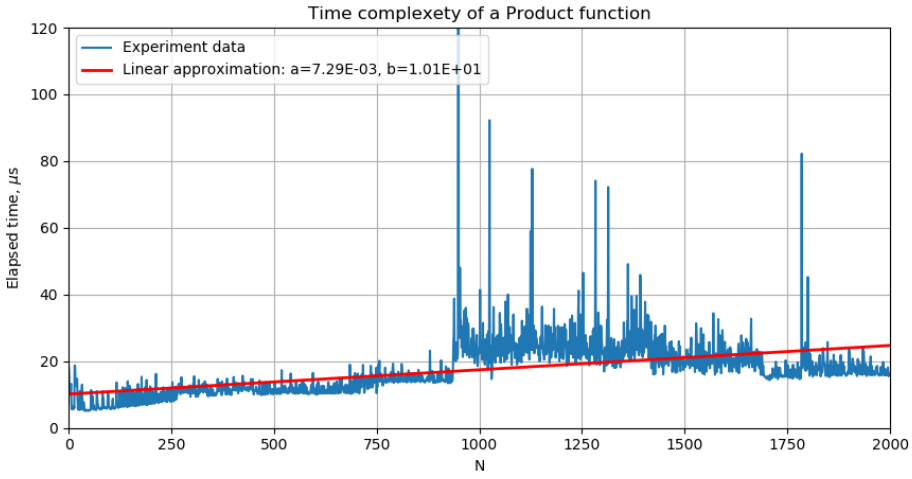

Figure 3: Approximation of experimental data of a product function

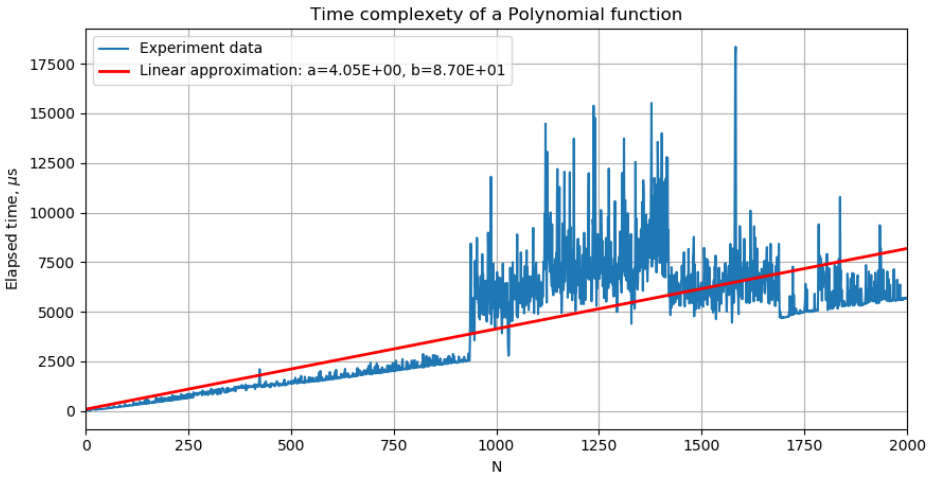

Figure 4: Approximation of experimental data of a direct polynomial calculation

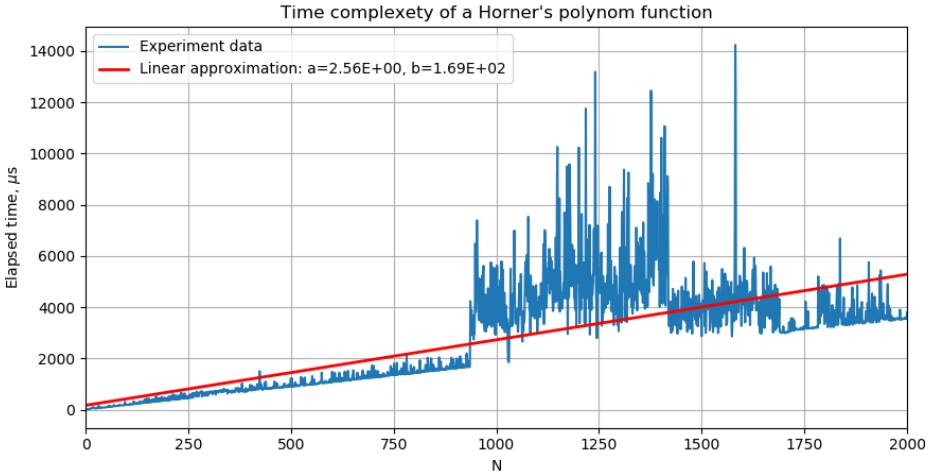

Figure 5: Approximation of experimental data of a Horner's polynomial calculation

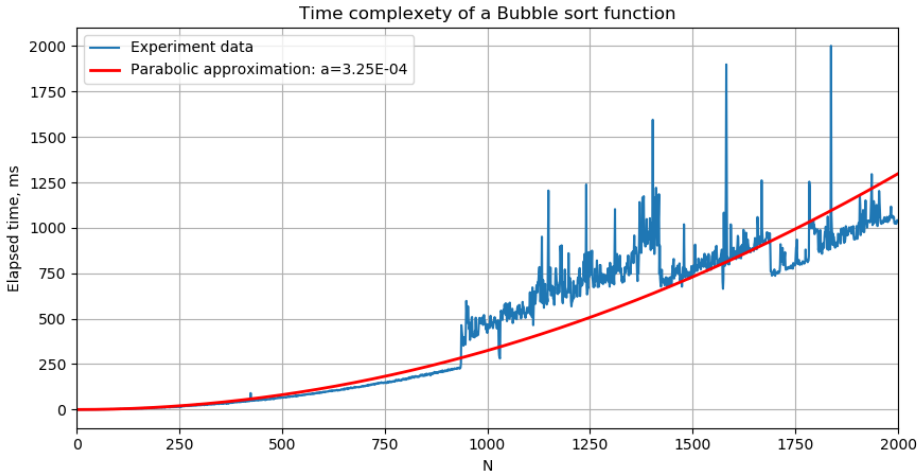

Figure 6: Approximation of experimental data of a Bubble sort

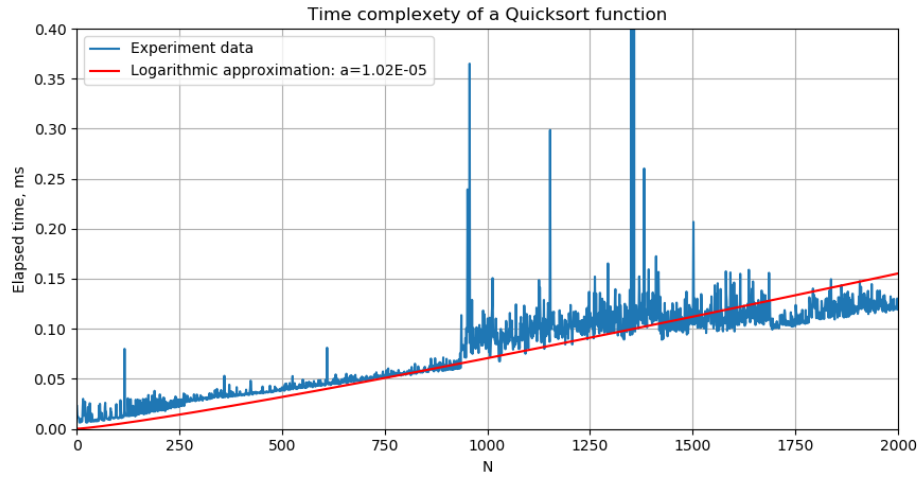

Figure 7: Approximation of experimental data of a Quicksort

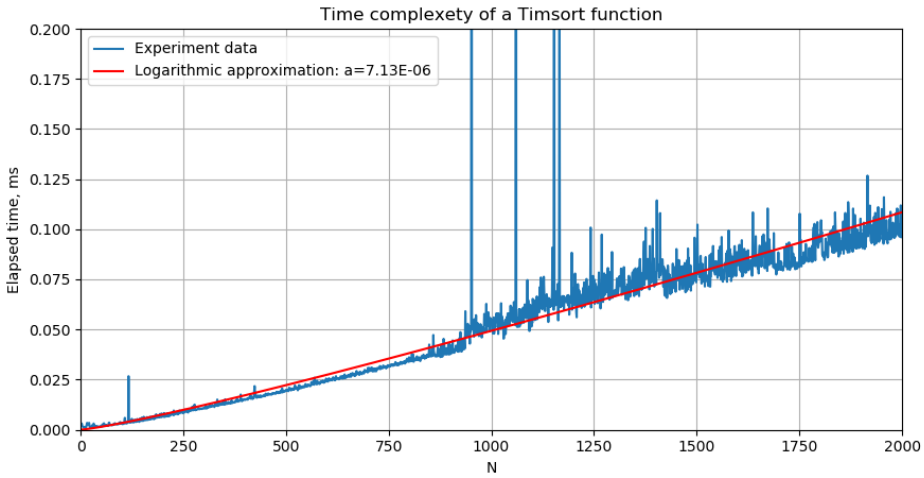

Figure 8: Approximation of experimental data of a Timsort

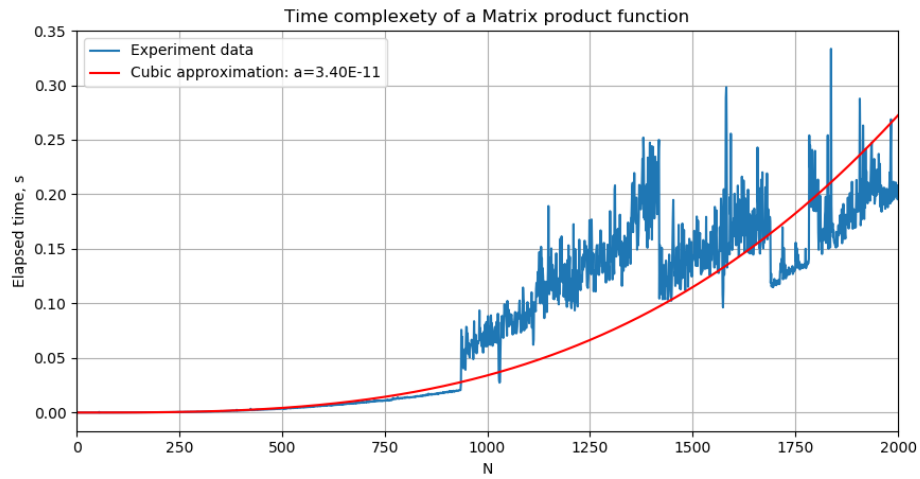

Figure 9: Approximation of experimental data of a matrix multiplication

From Figures 1, 2, 3, 4 and 5 that the experimental data from functions with expected linear time complexity are well approximated with the linear function.

Different rate of elapsed time in respect to number of inputs can be explained by implementation of the algorithms. For example, summation and product functions were taken from `numpy` module, which uses precompiled C executable to speedup certain computations, and on the other hand, polynomial computation was implemented using for-loop.

Bubble sort's experimental data is well approximated with parabola curve. The implementation of Bubble sort included two for-loops which gives $O(n^2)$ time complexity.

Quicksort (from `numpy`) and Timsort (default Python `sort()`) experimental data were approximated with *nlogn* function. Since time complexity of sorting algorithms depends on the input data, thus approximation on Figure 7 might does not look good, with increased number of samples it should converge to somewhat like *nlogn* curve.

For matrix multiplication `numpy.dot()` function was used. It is unclear which algorithm is used but the experimental data are well approximated with cubic curve (Fig. 9).

5 Conclusions

During this work the experimental data of time complexity of different functions was studies. Each function was ran five times and the mean elapsed time was considered on data with sizes from 0 to 2000.

Acquired data was approximated using linear function, parabola, *nlogn* and cubic curves based on the expected time complexity of the algorithm.

In section 4 the plots with experiment and approximate curves are present. From figures it can be seen that experimental data for each algorithm is well approximated with curves based on the expected time complexety.

Appendix

```
# Import all the needed modules
import numpy as np
from matplotlib import pyplot as plt
from scipy.optimize import curve_fit
import pandas as pd
from time import time
from copy import deepcopy as dp
import tqdm
from decimal import Decimal

# Setup pyplot's figure params
from pylab import rcParams
rcParams['figure.figsize'] = 10, 5
rcParams["figure.dpi"] = 100
img_dir = "./images/"

def const_func(v):
    v = v
```

```
    return v

def sum_func(v):
    return np.sum(v)

def prod_func(v):
    return np.prod(v)

def poly_simple(v, x=1.5):
    n = len(v)
    res = 0
    # Decimal from decimal module is used since 1.5**~1700
    # is bigger than float
    for p, k in zip(range(n), v):
        res += Decimal(x)**Decimal(p) * Decimal(k)

    return res

def poly_horner(v, x=1.5):
    n = len(v)
    res = 0
    # Decimal from decimal module is used since 1.5**~1700
    # is bigger than float
    for i in range(n-1, -1, -1):
        res = Decimal(res) * Decimal(x) + Decimal(v[i])

    return res

def bubble_sort(v):
    n = len(v)
    for i in range(1, n):
        swapped = False
        for j in range(n - i):
            if v[j] < v[j+1]:
                v[j], v[j+1] = v[j+1], v[j]
                swapped = True
        if not swapped:
            # If no swap was made - array is sorted
            break

    return v

def qucksort(v):
    return np.sort(v, kind="quick")

def timsort(v):
    return v.sort()

# Function takes list of samples and function and
# returns the elapsed time
def timeit(v, func):
    start = time()
    func(*v)
    elapsed_time = time() - start

    return elapsed_time

# Fucntion array with size length and random elements
# with uniformal distribution
def generate_v(length):
    return np.array([ np.random.uniform() for _ in range(length) ])

# Generate 2 random matrix with size n x n
```

```
def generate_matrixes(n):
    A = np.random.uniform(size=(n, n))
    B = np.random.uniform(size=(n, n))
    return A, B

def mult_matrix(A, B):
    return np.dot(A, B)

# Function collects average data for 5 runs
# for all function and returns array collected data
def run_test(v, A, B, n):
    elapsed_time = [0 for _ in range(10)]

    for _ in range(5):

        run = [n]

        run.append(timeit( [dp(v)], const_func ))
        run.append(timeit( [dp(v)], sum_func ))
        run.append(timeit( [dp(v)], prod_func ))
        run.append(timeit( [dp(v)], poly_simple ))
        run.append(timeit( [dp(v)], poly_horner ))
        run.append(timeit( [dp(v)], bubble_sort ))
        run.append(timeit( [dp(v)], qucksort ))
        run.append(timeit( [dp(v)], timsort ))
        run.append(timeit( [dp(A), dp(B)], mult_matrix ))

        elapsed_time = np.array(elapsed_time) +  np.array(run)

    elapsed_time = np.divide(elapsed_time, 5)

    return elapsed_time

# Function returns a pandas.DataFrame
# with avg execution time for each iteration
# for each function
def collect_data(n_max=2000):
    experiment_data = []

    for n in tqdm.tqdm_notebook(range(1, n_max+1)):
        v = generate_v(n)
        A, B = generate_matrixes(n)

        run_results = run_test(v, A, B, n)

        experiment_data.append(run_results)

    result = pd.DataFrame(data=experiment_data, columns=["n", "const",
                                                         "sum", "prod",
                                                         "simple_poly", "horner_poly",
                                                         "bubble_sort", "quicksort",
                                                         "timsort", "matrix_dot"])

    return result

# Constant function for approximation
def const_line(x, a):
    return [a] * len(x)

# Linear function for approximation
def lin_curve(x, k, b):
    return k*x + b

```

```
145 # Parabola function for approximation
146 def parabola(x, a, b):
147     return a*x**2
148 
149 # Cubic function for approximation
150 def cubic_curve(x, a):
151     return a*x**3
152 
153 # Log function for approximation
154 def log_curve(x, a):
155     return a*x*np.log(x)
156 
157 algos_time = collect_data(2000)
158 
159 # Following code plots experimental data
160 # approximates it and saves plot as a result
161 # Code repeats for all functions
162 # with corresponding approximation curves and
163 # small changes to the plot parameters
164 x_data = algos_time.n
165 fig, ax = plt.subplots()
166 plt.grid()
167 
168 plt.ylim(0, 12.5)
169 plt.xlim(0, 2000)
170 
171 ax.set_xlabel("N")
172 ax.set_ylabel("Elapsed time, $\mu$s")
173 ax.set_title(f"Time complexety of a Constant function")
174 
175 plt.plot(x_data, algos_time.const*10**6, label="Experiment data")
176 
177 popt, pcov = curve_fit(const_line, x_data, algos_time.const*10**6)
178 params = [ '%.2E' % Decimal(p) for p in popt ]
179 
180 plt.plot(x_data, const_line(x_data, *popt), 'r-',  linewidth=2,
181          label=f'Constant approximation: a={params[0]}, b={params[1]}') #%
    tuple(params))
182 
183 _ = plt.legend(loc="upper left")
184 
185 plt.savefig(img_dir + "const.png")
```