FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 1

“Experimental time complexity analysis”

Performed by

*Mariia Koroleva*

*J42322c*

Accepted by

Dr Petr Chunaev

St. Petersburg

2021

**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 with non-negative elements. For , implement the following calculations and algorithms:*

1. *(constant function);*
2. *(the sum of elements);*
3. *(the product of elements);*
4. *supposing that the elements of are the coefficients of a polynomial of degree , calculate the value by a direct calculation of (i.e. evaluating each term one by one) and by Horner’s method by representing the polynomial as  
   ;*
5. *Bubble Sort of the elements of ;*
6. *Quick Sort of the elements of ;*
7. *Timsort of the elements of .*

***II.*** *Generate random matrices and of size with non-negative elements. Find the usual matrix product for and .*

***III.*** *Describe the data structures and design techniques used within the algorithms.*

**Brief theoretical part**

The term “analysis of algorithms” usually means an investigation of algorithm’s efficiency with respect to two resources: running time and memory space (computational complexity).

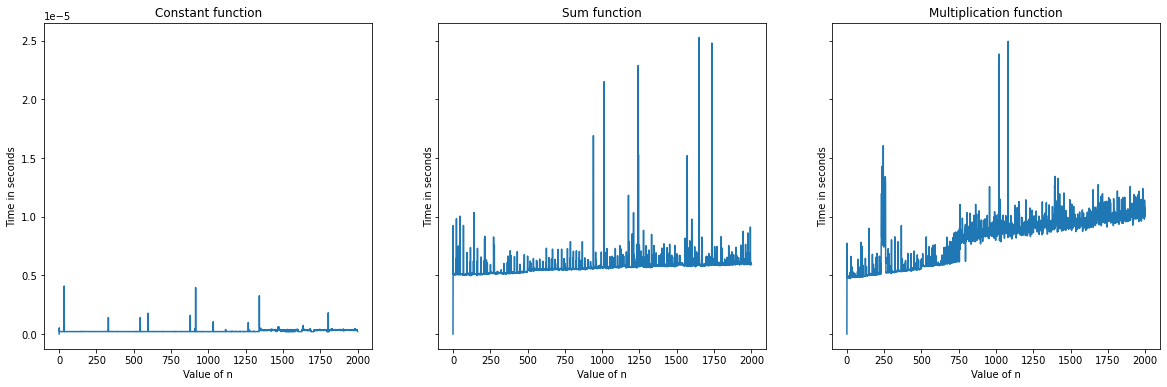
Running time complexity is a characteristic of an algorithm designed to give an idea of the amount of time it takes to run the algorithm on a given amount of data. The estimation of the time complexity of the algorithm is conducted by counting the number of elementary operations (additions, multiplications, etc.) performed by the algorithm for a given amount of data. It is assumed that the execution of each elementary operation requires a fixed amount of time.

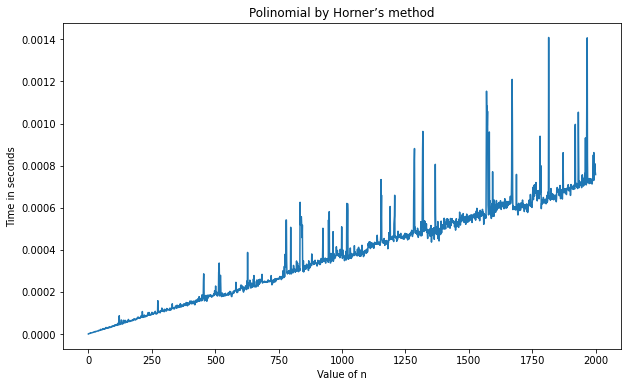
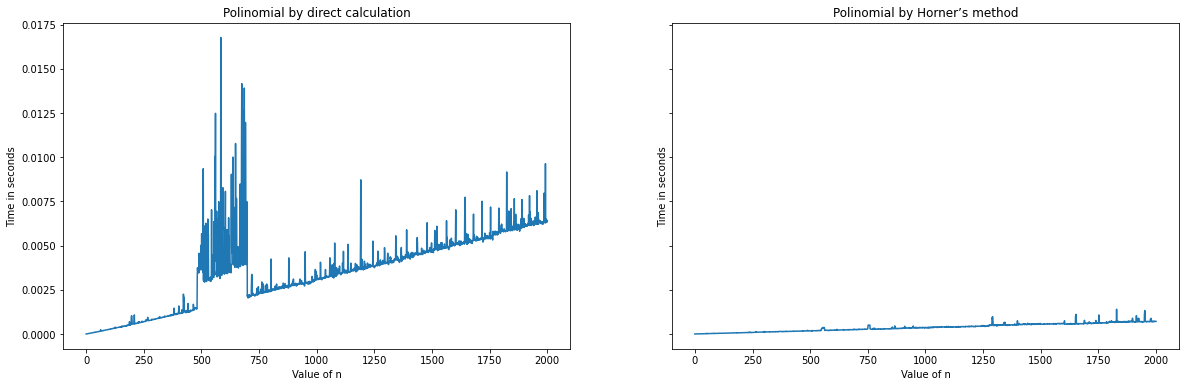
Since the running time of the algorithm can be different for data of varied sizes, it is customary to determine the time complexity using the running time function, T(n), which yields the time required to execute the algorithm of a problem of size n. We cannot determine this function exactly; it may contain unspecified constants.

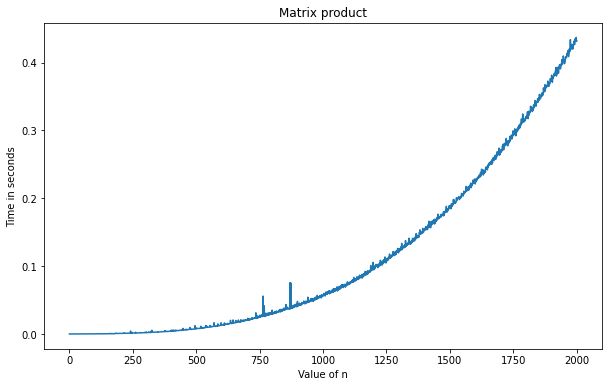
However, we can compare the efficiency of several algorithms by comparing their growth rates. The lower the growth rate, the faster the algorithm. In fact, the goal of the algorithm designer is to choose the algorithm with a grow rate of the running function as low as possible.

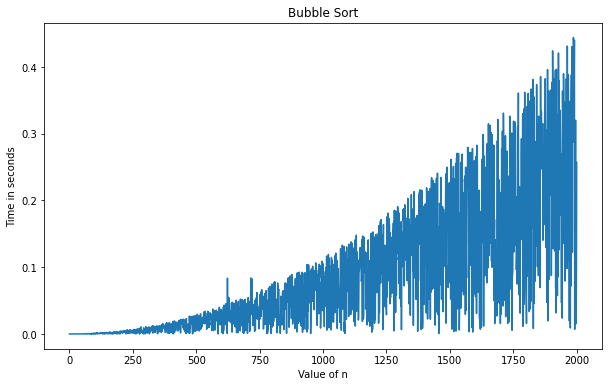
Estimating the time complexity of algorithms is not always straightforward; for some algorithms the time complexity is still unknown. In order to still get an idea of the time complexity of an algorithm, an empirical approach can be used. It consists in conducting a series of measurements of the running time of the algorithm when the volume of the input data (n-dimension) changes.

**Results**

All the algorithms were built by use of python3 functions (the notebook with codes is provided). A special time function was implemented for measuring the running time of any given algorithm by use of specially imported ‘time’ library. Then, the average time of 5 runs for each of n was counted. The matplotlib graphs of average time functions are the following:

** **

**Изображение выглядит как текст

Автоматически созданное описание**

**Conclusions**

As we see, watching at all the graphs, there is a number of data outliers. We can notice that since the actual measured time depends not only on the algorithm itself, but on the intensity of several parallel processes on the machine where that was implemented. However, mostly the trends are obvious:

1. The time function of constant function does not depend on the dimension of data as it is supposed to be theoretically.
2. Sum function is also rather stable, though is a bit more complex.
3. In production function we can see a noticeable growth.
4. The same is for polynomial, and both discussed methods of counting have similar efficiency.
5. Speaking of different methods of sorting, Timsort appears to be really one of the most efficient methods along with the Quicksort. The Bubble sort is the most complicated and time consuming.

**Appendix**

import numpy as np

import math

import random

import time

from statistics import mean

import matplotlib.pyplot as plt

from decimal import Decimal

# 1. Constant

func1 = lambda x: 1

# 2. Sum

func2 = lambda x: np.sum(x)

# 3. Multiplication

func3 = lambda x: np.prod(x)

# 4.1 Polinomial by direct calculation

def func4\_1(x):

    res = Decimal(0)

    for i in range(len(x) - 1):

      res += Decimal(x[i])\*(Decimal(1.5)\*\*i)

    return res

# 4.2 Polynomial by Horner's method

def func4\_2(x):

  res = x[-1]

  for i in range(-2, -len(x)-1, -1):

    res = x[i] + 1.5 \* res

  return res

# 5. Bubble Sort

def func5(x):

    while x[0] != min(x):

        n = len(x)

        for i in range(n-1):

          if x[i] > x[i+1]:

            x[i], x[i+1] = x[i+1], x[i]

    return x

# 6. Quick Sort

def func6(x):

  if len(x) <= 1:

        return x

  less = []

  eq = []

  gr = []

  a = random.choice(x)

  for i in range(len(x)):

    if x[i] > a:

      gr.append(x[i])

    elif x[i] == a:

      eq.append(x[i])

    else:

      less.append(x[i])

  return func6(less) + eq + func6(gr)

# 7. Timsort

# code is taken from https://www.geeksforgeeks.org/timsort/

# Python3 program to perform basic timSort

# Iterative Timsort function to sort the

# array[0...n-1] (like merge sort)

def func7(arr):

  MIN\_MERGE = 32

  def calcMinRun(n):

    """Returns the minimum length of a

    run from 23 - 64 so that

    the len(array)/minrun is less than or

    equal to a power of 2.

    e.g., 1=>1, ..., 63=>63, 64=>32, 65=>33,

    ..., 127=>64, 128=>32, ...

    """

    r = 0

    while n >= MIN\_MERGE:

      r |= n & 1

      n >>= 1

    return n + r

  # This function sorts array from left index to

  # to right index which is of size atmost RUN

  def insertionSort(arr, left, right):

    for i in range(left + 1, right + 1):

      j = i

      while j > left and arr[j] < arr[j - 1]:

        arr[j], arr[j - 1] = arr[j - 1], arr[j]

        j -= 1

  # Merge function merges the sorted runs

  def merge(arr, l, m, r):

    # original array is broken in two parts

    # left and right array

    len1, len2 = m - l + 1, r - m

    left, right = [], []

    for i in range(0, len1):

      left.append(arr[l + i])

    for i in range(0, len2):

      right.append(arr[m + 1 + i])

    i, j, k = 0, 0, l

    # after comparing, we merge those two arrays

    # in larger sub array

    while i < len1 and j < len2:

      if left[i] <= right[j]:

        arr[k] = left[i]

        i += 1

      else:

        arr[k] = right[j]

        j += 1

      k += 1

    # Copy remaining elements of left, if any

    while i < len1:

      arr[k] = left[i]

      k += 1

      i += 1

    # Copy remaining element of right, if any

    while j < len2:

      arr[k] = right[j]

      k += 1

      j += 1

  # Now the final part

  n = len(arr)

  minRun = calcMinRun(n)

  # Sort individual subarrays of size RUN

  for start in range(0, n, minRun):

    end = min(start + minRun - 1, n - 1)

    insertionSort(arr, start, end)

  # Start merging from size RUN (or 32). It will merge

  # to form size 64, then 128, 256 and so on ....

  size = minRun

  while size < n:

    # Pick starting point of left sub array. We

    # are going to merge arr[left..left+size-1]

    # and arr[left+size, left+2\*size-1]

    # After every merge, we increase left by 2\*size

    for left in range(0, n, 2 \* size):

      # Find ending point of left sub array

      # mid+1 is starting point of right sub array

      mid = min(n - 1, left + size - 1)

      right = min((left + 2 \* size - 1), (n - 1))

      # Merge sub array arr[left.....mid] &

      # arr[mid+1....right]

      if mid < right:

        merge(arr, left, mid, right)

      size = 2 \* size

  return arr

# II. Matrix product

def func\_II(A\_B):

  return A\_B[0].dot(A\_B[1])

runs = 5

algs = [func1, func2, func3, func4\_1, func4\_2, func5, func6, func7, func\_II]

algs\_title = ['Constant function', 'Sum function', 'Multiplication function',

              'Polinomial by direct calculation', 'Polinomial by Horner’s method',

              'Bubble Sort', 'Quick Sort', 'Timsort', 'Matrix product']

# Time Function

def time\_func(alg, data):

    start = time.perf\_counter()

    alg(data)

    return time.perf\_counter() - start

# Average time for 5 runs

def avg\_time(alg, n):

    '''

    n stands for the dimension of data

    '''

    if alg == func\_II:

        data = np.random.rand(n, n), np.random.rand(n, n)

    else:

        data = np.random.rand(n)

    time\_array = np.zeros(5)

    for i in range(5):

      time\_array[i] = time\_func(alg, data)

    return np.mean(time\_array)

# Making measures in n dimensions

def arr\_of\_measures(alg):

    n\_arr = np.array(list(range(1,2001)))

    res = np.zeros(len(n\_arr)  + 1)

    for i in range(len(n\_arr)):

        res[n\_arr[i]] = avg\_time(alg, n\_arr[i])

    return res

fig, axes = plt.subplots(1, 3, figsize=(20, 6), sharey=True)

for j, ax in zip([0, 1, 2], axes):

    ax.plot(arr\_of\_measures(algs[j]))

    ax.set\_title(algs\_title[j])

    ax.set\_xlabel('Value of n')

    ax.set\_ylabel('Time in seconds')

%%time

fig, ax = plt.subplots(figsize=(10, 6))

i = 3

ax.plot(arr\_of\_measures(algs[i]))

ax.set\_title(algs\_title[i])

ax.set\_xlabel('Value of n')

ax.set\_ylabel('Time in seconds')

%%time

fig, ax = plt.subplots(figsize=(10, 6))

i = 4

ax.plot(arr\_of\_measures(algs[i]))

ax.set\_title(algs\_title[i])

ax.set\_xlabel('Value of n')

ax.set\_ylabel('Time in seconds')

%%time

fig, ax = plt.subplots(figsize=(10, 6))

i = 5

ax.plot(arr\_of\_measures(algs[i]))

ax.set\_title(algs\_title[i])

ax.set\_xlabel('Value of n')

ax.set\_ylabel('Time in seconds')

%%time

fig, axes = plt.subplots(1, 2, figsize=(20, 6), sharey=True)

for j, ax in zip([6, 7], axes):

    ax.plot(arr\_of\_measures(algs[j]))

    ax.set\_title(algs\_title[j])

    ax.set\_xlabel('Value of n')

    ax.set\_ylabel('Time in seconds')

%%time

fig, ax = plt.subplots(figsize=(10, 6))

i = -1

ax.plot(arr\_of\_measures(algs[i]))

ax.set\_title(algs\_title[i])

ax.set\_xlabel('Value of n')

ax.set\_ylabel('Time in seconds')