FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 1

“Experimental time complexity analysis”

Performed by

Zakhar Pinaev

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2021

Goal

Experimental study of the time complexity of different algorithms.

Problems and methods

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.

Brief theoretical part

The importance of analyzing the complexity of algorithms is caused by the need to efficiently use computational resources. When talking about the efficiency of an algorithm, it is necessary to consider two aspects: running time and memory space, but in this work, all attention will be paid to the time.

In computer science, there are best, worst, and average cases of a given algorithm, which express what the resource usage is at least, at most and on average, respectively. Best case is the function which performs the minimum number of steps, worst case - maximum number of steps, and average case - average number of steps on input data of n elements. In this paper, the average case will be used as an estimate of the execution time.

A running time function, T(n), yields the time required to execute the algorithm of a problem of size n. A running time function, T(n), yields the time required to execute the algorithm of a problem of size n. For example, $T(n) = an^2 + bn + c$ has growth rate O(n2).

Results

I. First, an *n*-dimensional random vector *v* with non-negative elements was generated. After that, the following calculations and algorithms were implemented for *v* (for each *n* from 1 to 2000 were taken 5 runs for each function, described below, to measure the average computer execution time):

1) *f*(***v***) = *const* (*constant function*)

From a theoretical point of view, the time complexity of a constant function can be described as O(1), since the execution time of a constant function does not depend on the size of the input data. As can be seen from Figure 1, during experiments the average execution time of a constant function could be estimated as O(1), so that the theoretical and empirical time complexities are the same.

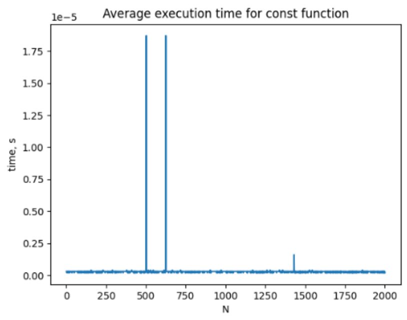

Figure 1 – the average computer execution time of constant function, where N – size of the input vector

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

The theoretical time complexity of the sum of the elements is O(n) because it is needed to look at each item in the vector ***v*** to add them up. As can be seen from Figure 2, during experiments the average execution time of a sum function could be estimated as O(n), so that the theoretical and empirical time complexities are the same.

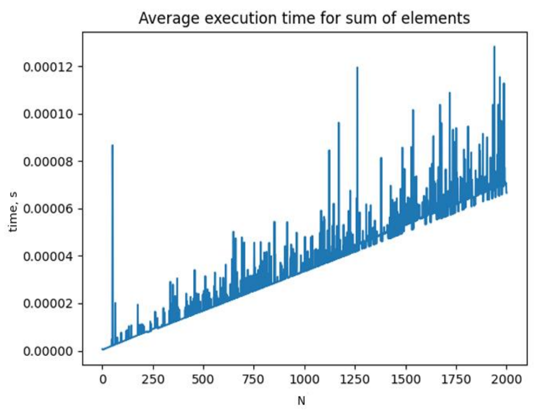

Figure 2 – the average computer execution time of sum function, where N – size of the input vector

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

The theoretical time complexity of the product of the elements is O(n) because (same as with sum function) it is needed to traverse through the array one time to get the result. As can be seen from Figure 3, during experiments the average execution time of a product function is close to be estimated as O(n), but it seems, like it grows a little faster (for example, compared to the graph of the time complexity of the sum function in Figure 2). This may be since the multiplication operation is more computationally expensive in comparison with the addition operation. Thus, theoretical and empirical time complexities are close to each other.

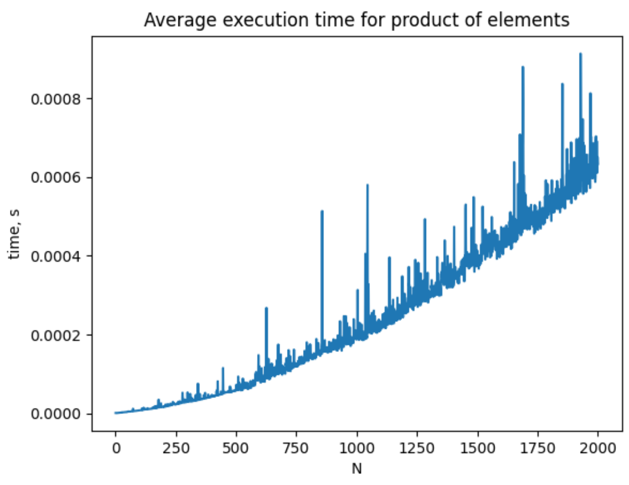

Figure 3 – the average computer execution time of product function, where N – size of the input vector

4.1) $f(\boldsymbol{v}) = \sum_{k=1}^{n} v_k x^{k-1}$ (*polynomial direct calculation*)

The theoretical time complexity of the direct calculation of polynomial (Brute Force method) is O(n^2) and might be improved up to O(n *log(*n)). As can be seen from Figure 4, during experiments the average execution time of the direct polynomial calculation grows a little faster, than O(n), so that it can be estimated as O(n *log(*n)), so that the theoretical and empirical time complexities are the same.

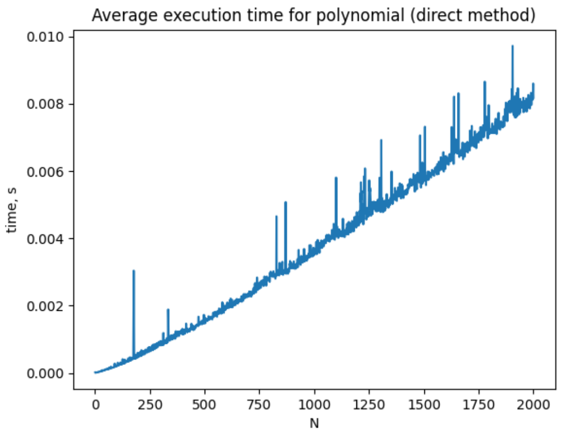

Figure 4 – the average computer execution time of direct polynomial calculation function, where N – size of the input vector

4.2) $f(\boldsymbol{v}) = v_1 + x(v_2 + x(v_3 + \cdots))$ (*polynomial calculation using Horner's method*)

The theoretical time complexity of the calculation of polynomial using Horner's method is O(n). As can be seen from Figure 5, during experiments the average execution time of Horner's method of calculation the polynomial, could be estimated as O(n), so that the theoretical and empirical time complexities are the same.

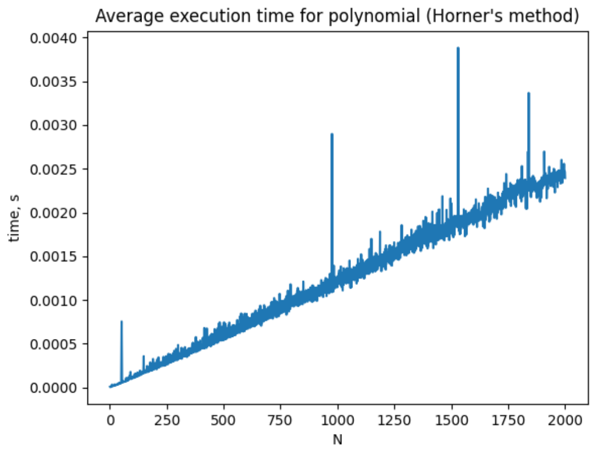

Figure 5 – the average computer execution time of polynomial calculation using Horner's method, where N – size of the input vector

5) *Bubble Sort of the elements of **v***

The theoretical time complexity of the Bubble Sort is O(n^2). As can be seen from Figure 6, during experiments the average execution time of Bubble Sort could be estimated as O(n^2), so that the theoretical and empirical time complexities are the same.

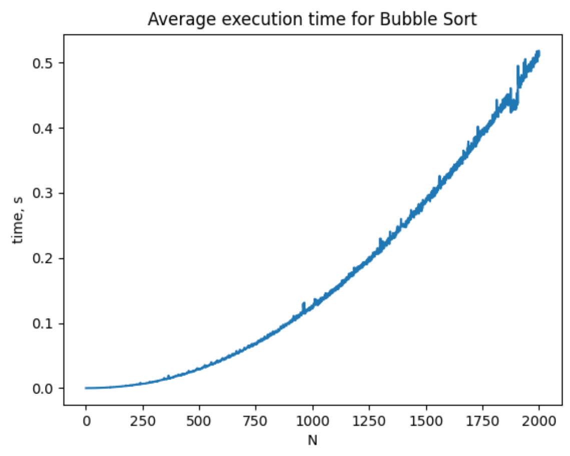

Figure 6 – the average computer execution time of Bubble Sort, where N – size of the input vector

6) *Quick Sort of the elements of **v***

The theoretical time complexity of the Quick Sort is O(n *log*(n)). As can be seen from Figure 7, during experiments the average execution time of Quick Sort could be estimated as O(n *log*(n)), since it grows a little faster, than O(n), so that the theoretical and empirical time complexities are the same.

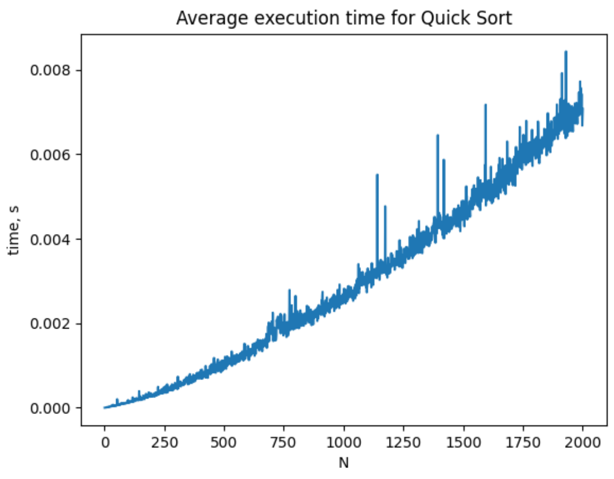

Figure 7 – the average computer execution time of Quick Sort, where N – size of the input vector

7) *Timsort of the elements of **v***

The theoretical time complexity of the Timsort is O(n *log*(n)). As can be seen from Figure 8, during experiments the average execution time of Timsort could be estimated as O(n *log*(n)), since it grows a little faster, than O(n), so that the theoretical and empirical time complexities are the same.

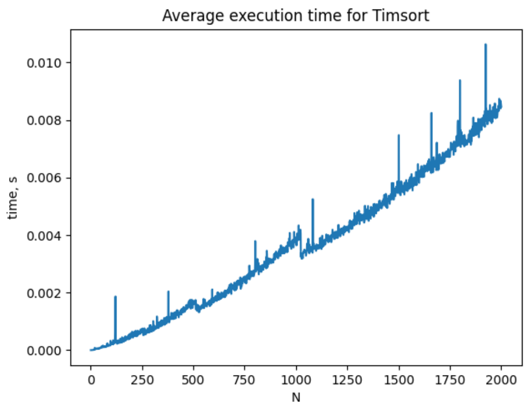

Figure 8 – the average computer execution time of Timsort, where N – size of the input vector

II. Then, random matrices A and B of size *n*x*n* with non-negative elements were generated. The theoretical time complexity of the product of matrices is O(n^3). It was not possible to compute matrices of size 2000x2000, so in this work maximum size of matrix was *n* = 250. As can be seen from Figure 9, during experiments the average execution time of product of the matrices could be estimated as O(n^3), since it grows a much faster, than O(n^2) which can be seem after comparing Figure 9 with time complexity of Bubble Sort (Figure 6), so that the theoretical and empirical time complexities are the same.

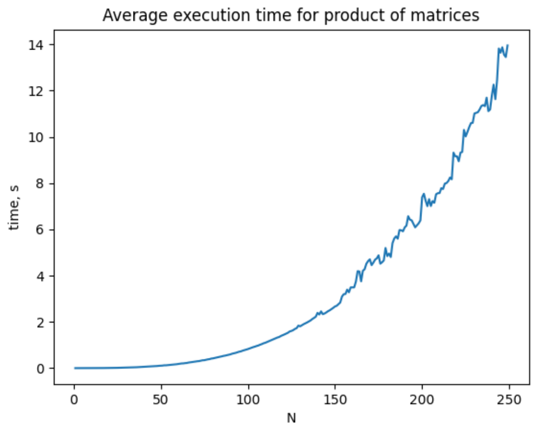

Figure 8 – the average computer execution time of matrices product, where N – size of the input matrices (*n*x*n*)

III. As a constant function was considered the function, which allows to find out if the first element of the vector is even. For each function considered, a separate implementation was written to avoid inaccuracies associated with optimizing built-in functions. The direct calculation of polynomial could be considered as Brute Force method. For testing each algorithm, python list was used, except for the second step with matrix multiplication, which used numpy arrays.

Conclusion

As a result of the work, various functions were implemented, for each of which an assessment of the time complexity was carried out. For constant function, sum, direct polynomial computation, Horner polynomial computation, bubble sort, quick sort, timsort, and matrix multiplication, the corresponding theoretical and empirical time complexities coincided. For the rest of the functions, these estimates were very close to each other. In addition, the analysis of time complexities was hampered by outliers on the graphs, which may be due to processes occurring on the computer in parallel with the calculations.

Appendix

```
import random
import matplotlib.pyplot as plt
import decimal
import numpy as np
from scipy.interpolate import interp1d
import timeit

decimal.getcontext().prec = 100
MIN_MERGE = 32

def is_even(myList, index):
    return myList[index] % 2 == 0

def sum_func(mylist):   #list is passed to the function
    summ = 0
    for n in mylist:
        summ += n
    return summ

def multiplyList(myList):
    # Multiply elements one by one
    result = 1
    for x in myList:
        result = result * x
    return result


def polynom_direct(myList, x):
    poly_sum = decimal.Decimal(0)
    for k in range(len(myList)):
        poly_sum = decimal.Decimal(poly_sum) + decimal.Decimal(myList[k]) *
decimal.Decimal((decimal.Decimal(x)**decimal.Decimal(k)))
    return decimal.Decimal(poly_sum)


def polynom_horner(myList, x):
    poly_sum = decimal.Decimal(0)
    for k in reversed(myList):
        poly_sum = poly_sum * decimal.Decimal(x) + decimal.Decimal(k)
    return decimal.Decimal(poly_sum)


def bubble_sort(our_list):
    for i in range(len(our_list)):
        for j in range(len(our_list) - 1):
            if our_list[j] > our_list[j+1]:
                our_list[j], our_list[j+1] = our_list[j+1], our_list[j]


def partition(arr, low, high):
    i = (low - 1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
```

```python
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)


def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)


def calcMinRun(n):
    r = 0
    while n >= MIN_MERGE:
        r |= n & 1
        n >>= 1
    return n + r


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


def merge(arr, l, m, r):
    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
    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
    while i < len1:
        arr[k] = left[i]
        k += 1
        i += 1
    while j < len2:
        arr[k] = right[j]
        k += 1
        j += 1


def timSort(arr):
    n = len(arr)
    minRun = calcMinRun(n)
    for start in range(0, n, minRun):
        end = min(start + minRun - 1, n - 1)
        insertionSort(arr, start, end)
    size = minRun
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), (n - 1))
            if mid < right:
                merge(arr, left, mid, right)
        size = 2 * size


def matrix_product(A, B, result):
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]
    return result


N = list(range(1, 2001))

time_1 = [0] * len(N)
time_2 = [0] * len(N)
time_3 = [0] * len(N)
time_4_1 = [0] * len(N)
time_4_2 = [0] * len(N)
time_5 = [0] * len(N)
time_6 = [0] * len(N)
time_7 = [0] * len(N)
time_8 = [0] * len(N)

for n in N:
    v = [random.randrange(1, 100, 1) for r in range(n)]
    A = np.random.randint(10, size=(n, n))
    B = np.random.randint(10, size=(n, n))
    for k in range(1, 6):


        # point 1.1) - Constant function
        start = timeit.default_timer()
        temp = is_even(v, 0)
        time_1[n - 1] = time_1[n - 1] + (timeit.default_timer() - start) / 5



        # point 1.2) - the sum of elements v
        start = timeit.default_timer()
        temp = sum_func(v)
        time_2[n - 1] = time_2[n - 1] + (timeit.default_timer() - start) / 5
        #time_2[n - 1] = time_2[n - 1] + (time.time() - start) / 5



        # point 1.3) - the product of elements v
        start = timeit.default_timer()
        temp = multiplyList(v)
        time_3[n - 1] = time_3[n - 1] + (timeit.default_timer() - start) / 5



        # point 1.4.1) - the polynom for v direct method
        start = timeit.default_timer()
        temp = polynom_direct(v, 1.5)
        time_4_1[n - 1] = time_4_1[n - 1] + (timeit.default_timer() - start)
/ 5
```

```
        # point 1.4.2) - the polynom for v Horner's method
        start = timeit.default_timer()
        temp = polynom_horner(v, 1.5)
        time_4_2[n - 1] = time_4_2[n - 1] + (timeit.default_timer() - start)
/ 5



        # point 1.5) - the bubble sort of v
        v1 = v.copy()
        start = timeit.default_timer()
        bubble_sort(v1)
        time_5[n - 1] = time_5[n - 1] + (timeit.default_timer() - start) / 5



        # point 1.6) - the Quick sort of v
        v1 = v.copy()
        start = timeit.default_timer()
        quickSort(v1, 0, n - 1)
        time_6[n - 1] = time_6[n - 1] + (timeit.default_timer() - start) / 5



        # point 1.7) - the Timsort of v
        v1 = v.copy()
        start = timeit.default_timer()
        timSort(v1)
        time_7[n - 1] = time_7[n - 1] + (timeit.default_timer() - start) / 5


        #if n > 50:
        #   continue
        #point 2) - the product of matrices A and B
        result = np.zeros((n, n), dtype=int)
        start = time.time()
        temp = matrix_product(A, B, result)
        time_8[n - 1] = time_8[n - 1] + (time.time() - start) / 5



"""
# graph for point 1.1
plt.figure(1)
plt.plot(N, time_1)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for const function')



# graph for point 1.2
plt.figure(2)
#p1 = np.polyfit(N, time_2, 1)
#y_fit = np.polyval(p1, time_2)
#time_interp = interp1d(N, y_fit, kind='linear')
plt.plot(N, time_2)
#plt.legend(['data', 'linear interp'], loc='best')
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for sum of elements')
```

```
# graph for point 1.3
plt.figure(3)
plt.plot(N, time_3)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for product of elements')



# graph for point 1.4.1
plt.figure(4)
plt.plot(N, time_4_1)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for polynomial (direct method)')


# graph for point 1.4.2
plt.figure(5)
plt.plot(N, time_4_2)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for polynomial (Horner\'s method)')



# graph for point 1.5
plt.figure(6)
plt.plot(N, time_5)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for Bubble Sort')



# graph for point 1.6
plt.figure(7)
plt.plot(N, time_6)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for Quick Sort')



# graph for point 1.7
plt.figure(8)
plt.plot(N, time_7)
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for Timsort')



# graph for point 2
plt.figure(9)
plt.plot(N, time_8)
#plt.xlim([0, 50])
plt.xlabel('N')
plt.ylabel('time, s')
plt.title('Average execution time for product of matrices')
"""
plt.show()
```