FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 1

“Experimental time complexity analysis”

Performed by

Victor Voronin

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2020

**Goal**

Experimental study of the time complexity of different algorithms

**Formulation of the problem**

1. Generate an n-dimensional random vector with non-negative elements. For 𝒗, 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 𝒗 are the coefficients of a polynomial 𝑃 of degree n − 1, calculate the value 𝑃(1.5) by a direct calculation of

(i.e. evaluating each term one by one) and by Horner’s method by representing the polynomial as

;

* 1. Bubble Sort of the elements of 𝒗;
  2. Quick Sort of the elements of 𝒗;
  3. Timsort of the elements of 𝒗.

1. Generate random matrices A and B of size n × n with non-negative elements. Find the usual matrix product for A and B.
2. Describe the data structures and design techniques used within the algorithms.

**Brief theoretical part**

To solve the task it is supposed to use the following standard libraries:

* library random to generate random n-dimensional vector
* library numpy to generate random matrixes
* library time for measurement time of execution of code
* matplotlib.pyplot to create graphs
* library math to calculate logarithms

Big-O notation

Let f(n) and g(n) be functions mapping positive integers to positive real numbers. We say that f(n) is order of O(g(n)) if there is a real constant c > 0 and an integer constant n0 ≥ 1 such that

Horner’s method

Given the polynomial

where {\displaystyle a\_{0},\ldots ,a\_{n}} are constant coefficients, the problem is to evaluate the polynomial at a specific value {\displaystyle x\_{0}}. {\displaystyle x.}

For this, a new sequence of constants is defined recursively as follows:

Then is value of and polinomial can be written in the form:

Horner's method requires only n additions and n multiplications. This allows Horner method to calculate value of polinomial function as fast as naive algorithm does.

Bubble sort

This is 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.

Complicity time is O(n) for best case and O(n2) for worst average cases.

Quick sort

The quick-sort algorithm sorts a sequence S using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps:

1. Divide: If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the pivot. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
   1. L, storing the elements in S less than x
   2. E, storing the elements in S equal to x
   3. G, storing the elements in S greater than x

Of course, if the elements of S are distinct, then E holds just one element—the pivot itself.

1. Conquer: Recursively sort sequences L and G.
2. Combine: Putback the elements intoS inorder by first inserting the elements of L, then those of E, and finally those of G.

Complicity time is O(nlog(n)) for best and average cases and O(n2) for worst case.

Timsort

Terms

* N is the size of the input array
* Run is an ordered subarray in the input array. Moreover, it is ordered either loosely in ascending order or strictly descending order.
* Minrun is the minimum subarray size by which the original array will be divided.

Computing minrun

1. The minrun number (the minimum size of an ordered sequence) is determined based on N based on the following principles: it should not be too large, since insertion sort will be applied to the minrun size subarray in the future, and it is effective only on small arrays.
2. It should not be too small, since the smaller the subarray, the more iterations of merging subarrays will have to be performed at the last step of the algorithm. The optimal value for N / minrun is a power of 2 (or close to it). This requirement is due to the fact that the subarray merge algorithm works most efficiently on subarrays of approximately equal size.

Splitting into subarrays and sorting them.

1. The current element pointer is placed at the beginning of the input array.
2. Starting from the current element, this array is searched for the ordered subarray run. By definition, run will unambiguously include the current element and the next one after it. If the resulting subarray is ordered in descending order, the elements are rearranged so that they go in ascending order.
3. If the size of the current run is less than minrun, the elements following the found run are selected in the amount of minrun-size (run). Thus, the output will be a subarray of minrun size or more, part of which (ideally, all of it) is ordered.
4. Insert sort is applied to this subarray. Since the size of the subarray is small and part of it is already ordered, sorting works quickly and efficiently.
5. The current element pointer is placed on the element following the subarray.
6. If the end of the input array is not reached - go to step 2, otherwise - the end of this step.

Merge

1. An empty stack of pairs <subarray start index> - <subarray size> is created. The first ordered subarray is taken.
2. The data pair <start index> - <size> for the current subarray is added to the stack.
3. Determines whether the current subarray should be merged with the previous ones. For this, two rules are checked (let X, Y and Z be the sizes of the three top subarrays in the stack):
   * + 1. Z> Y + X
       2. Y> X
4. If one of the rules is violated, the Y array is merged with the smaller of the X and Z arrays. It is repeated until both rules are satisfied or the data is completely ordered.
5. If there are still not considered subarrays - take the next one and go to step 2. Otherwise - the end.

Merge of subarrays, where it requare

1. A temporary array is created in the size of the smaller of the joined subarrays.
2. The smallest of the subarrays is copied into a temporary array
3. Pointers of the current position are placed on the first (last) elements of the larger and temporary array.
4. At each next step, the value of the current elements in the larger and temporary arrays is considered, the smaller (larger) of them is taken and copied into a new sorted array. The current element pointer moves in the array from which the element was taken.
5. Point 4 is repeated until one of the arrays ends.
6. All elements of the remaining array are added to the end of the new array.

Complicity time is O(n) for best case and O(nlog(n)) for worst average cases.

**Results**

1. The measurement of the running time of the algorithm and plotting graphs were carried out using method get\_plot(test\_function). For instance, get\_plot(sum\_function). For more details see Appendix 1.
2. Time of calculating function (constant function) doesn’t depend on size of the generated vector. Theoretical time complexities is O(1). The function can be approximated by a constant function of the form . The plot of computation time of versus n is shown in Figure 1. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 2.

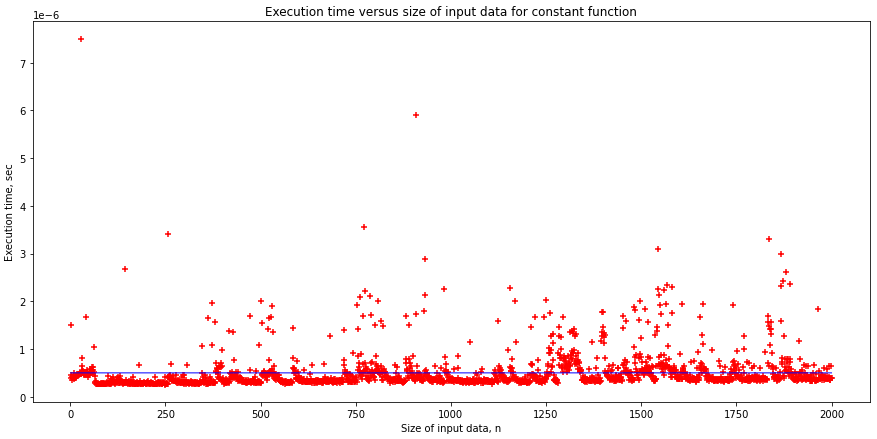


Figure 1 Execution time versus size of input data for constant function

1. Time of calculating function (the sum of elements) linearly depend on size of the generated vector. Each step of algorithm has operation sum that perform in constant time and repeat n time. Theoretical time complexities is O(n). The function can be approximated by a linear function of the form . The plot of computation time of versus n is shown in Figure 2. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 3.

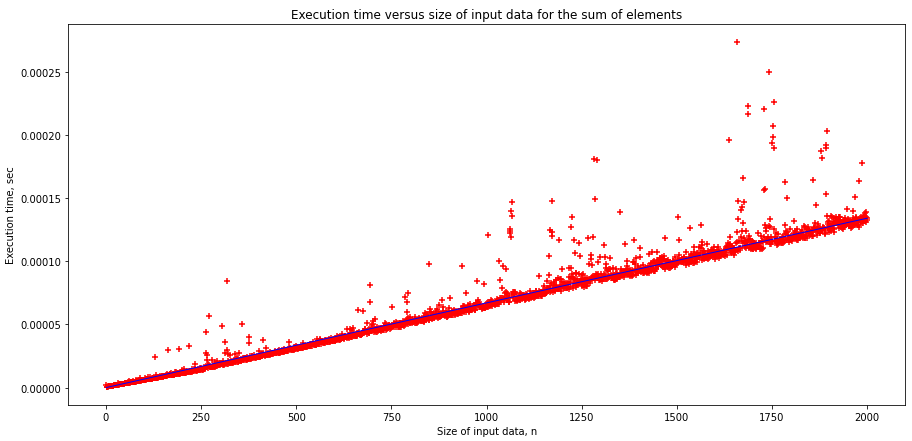


Figure 2 Execution time versus size of input data for sum elements

1. Time of calculating function (the product of elements) quadratic dependence on the number of bits in the multiplied variables. The function can be approximated by a polinomial function of the form , where a, b, c are some real numbers. The plot of computation time of versus n is shown in Figure 3. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 4.

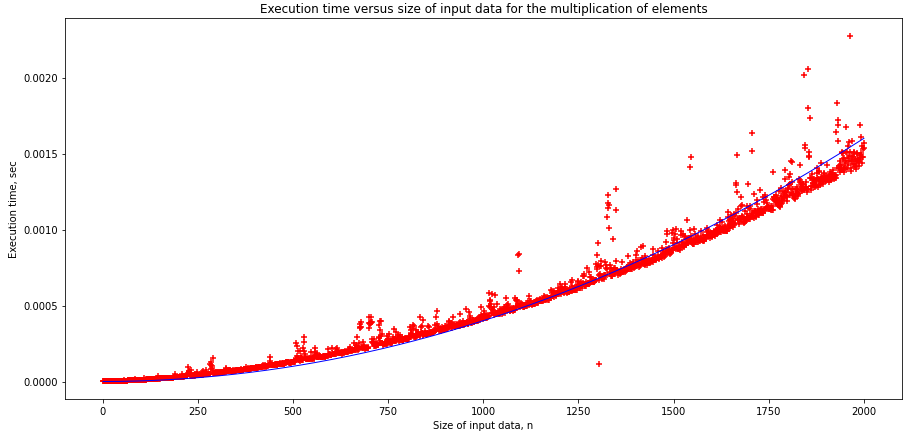


Figure 3 Execution time versus size of input data for multiplication elements

1. Time of calculating polinomial function with naive method by representing the polynomial as linearly depend on size of the generated vector. Each step of algorithm has operation multiplicate and sum that perform in constant time and repeat n time. Theoretical time complexities is O(n). The function can be approximated by a linear function of the form . The plot of computation time of versus n is shown in Figure 4. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 5.

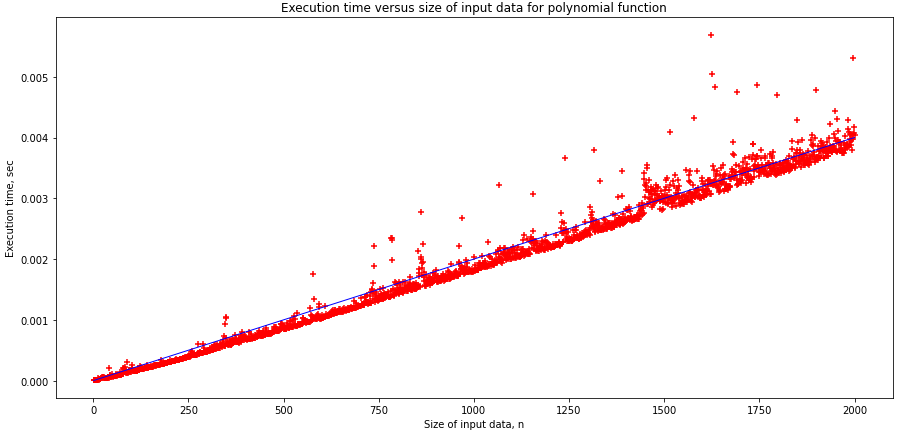


Figure 4 Execution time versus size of input data for polinomial function (naive method)

1. Time of calculating polinomial function with Horner method by representing the polynomial as linearly depend on size of the generated vector. Each step of algorithm has operation multiplicate and sum that perform in constant time and repeat n time. Theoretical time complexities is O(n). The function can be approximated by a linear function of the form . The plot of computation time of versus n is shown in Figure 5. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 6.

Unlike naive algorithm, Horner's method requires only n additions and n multiplications. This allows Horner method to calculate value of polinomial function about twice as fast as naive algorithm does.

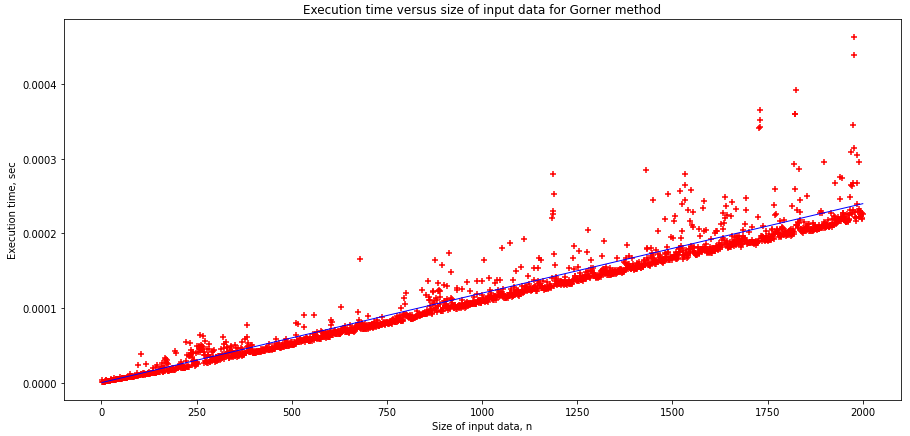


Figure 5 Execution time versus size of input data for polinomial function (Horner method)

1. Bubble sort has complixity time O(n2) in average and worst cases. The function of time sorting can be approximated by a polinomial function of the form , where a, b, c are some real numbers. The plot of sorting time of elements of vector 𝒗 versus its size n is shown in Figure 6. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 7.

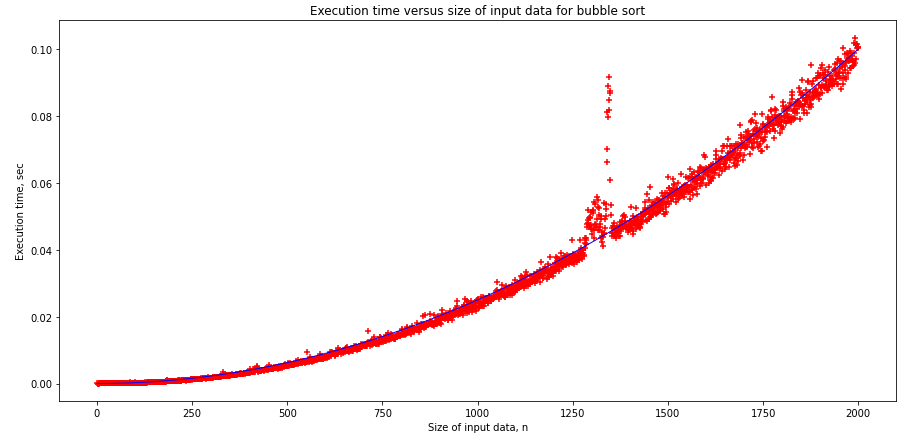


Figure 6 Sorting time elements of vector 𝒗 with bubble sort

1. Quick sort has complixity time O(nlogn) in average and O(n2) in worst case. The function of time sorting can be approximated by a logarithmic function of the form . The plot of sorting time of elements of vector 𝒗 versus its size n is shown in Figure 7. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 8.

Unlike bubble sort, quick sort uses a data structure called multiway search tree. This allows increse speed of working algorithm.

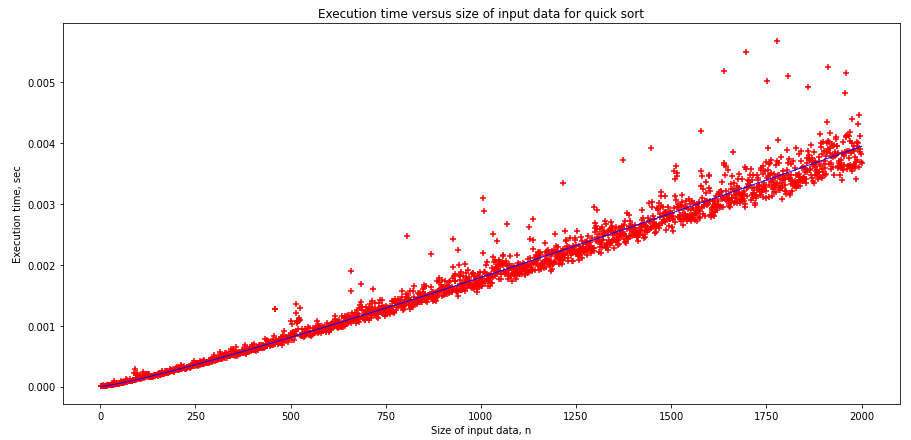
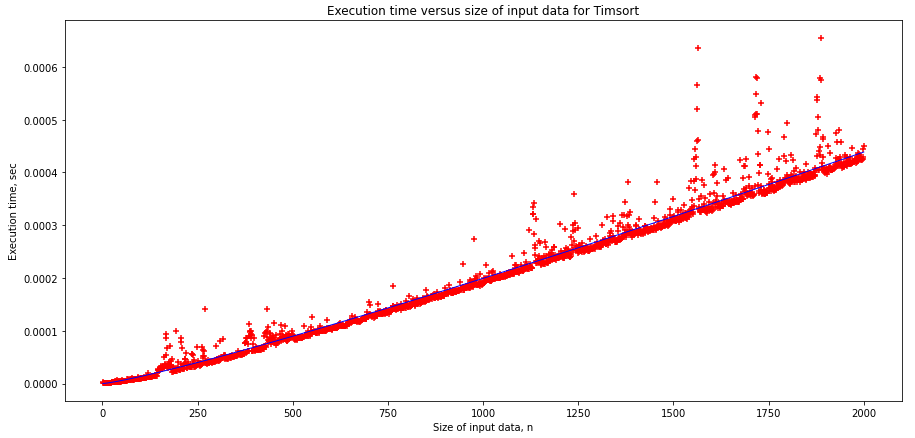


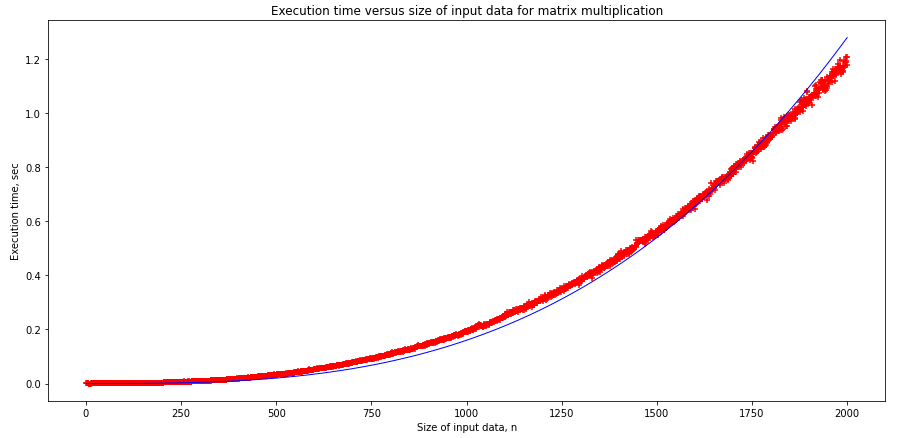
Figure 7 Sorting time elements of vector 𝒗 with quick sort

1. Timsort has complixity time O(nlogn) in average and worst cases. This is default way of sort for functions list.sort() and sorted() from Python 2.3. The function of time sorting can be approximated by a logarithmic function of the form . The plot of sorting time of elements of vector 𝒗 versus its size n is shown in Figure 8. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 9.

Acceleration of work is achieved by using of partial ordering of some parts of the input array.



1. Time complexities of function where A and B are random matrixes size of n x n is O(n3). Mupltiplication martixes algorithm has three loop “for”. Each step of algorithm has operation multiplicate corresponding elements from matrixes A and B and increse index of loops that perform in constant time. The function can be approximated by a polinomial function of the form , where a, b, c and d are some real numbers The plot of computation time of versus n is shown in Figure 9. Blue line shows approximity function, scattered red pluses ‒ empirical complexities. For code see Appendix 10.



**Conclusions**

During the execution of the task, graphs of the empirical and theoretical dependence of the execution time of the algorithm on the size of the input data were obtained.

Appendix 1

def get\_plot(test\_function):

  n = list(range(1, 2001))

  elapsed\_time = []

  theoretical\_time = []

  for i in n:

    vector = random.sample(range(1000000), k=i)

    t\_start = time.process\_time()

    for j in range (5):

      test\_function(vector)

    elapsed\_time.append((time.process\_time() - t\_start) / 5)

    theoretical\_time.append(0.00000000016 \* i \*\* 3) #approximity function

is rewrite for each test function by hand

  plt.figure(figsize=(15,7))

  plt.title('Execution time versus size of input data for matrix multiplication')

  plt.xlabel('Size of input data, n')

  plt.ylabel('Execution time, sec')

  plt.scatter(n, elapsed\_time, marker="+", color='red')

  plt.plot(n, theoretical\_time, linewidth = 1, color='blue')

  plt.show()

Appendix 2

def constant\_function(vector):

  function\_value = 1

Appendix 3

def sum\_function(vector):

  function\_value = 0

  for i in range(len(vector)):

    function\_value += vector[i]

Appendix 4

def mult\_function(vector):

  function\_value = 1

  for i in range(len(vector)):

     function\_value = function\_value \* vector[i]

Appendix 5

def polynom\_function(vector):

  x = decimal.Decimal(1.5) #use lib decimal to avoid overflow

  temp = 0

  for i in range(len(vector)):

    temp += decimal.Decimal(vector[i]) \* decimal.Decimal(x) \*\* decimal.Decimal((i))

Appendix 6

def Horner\_method(vector):

  x = 1.5

  temp = vector[-1]

  for i in reversed(range(1, len(vector))):

    temp1 = vector[i-1] + x \* temp

Appendix 7

def bubble\_sort(vector):

  still\_swapping = True

  while still\_swapping:

      still\_swapping = False

      for i in range(len(vector) - 1):

          if vector[i] > vector[i+1]:

              vector[i], vector[i+1] = vector[i+1], vector[i]

              still\_swapping = True

Appendix 8

def quick\_sort(vector):

  less\_pivot = []

  equal\_pivot = []

  greater\_pivot = []

  if len(vector) > 1:

    pivot = vector[0]

    for x in vector:

      if x < pivot:

          less\_pivot.append(x)

      elif x == pivot:

          equal\_pivot.append(x)

      elif x > pivot:

          greater\_pivot.append(x)

    return quick\_sort(less\_pivot) + equal\_pivot + quick\_sort(greater\_pivot)

  else:

      return vector

Appendix 9

def timsort (vector):

  sorted(vector) #sorted() uses timsort from Python 2.3

Appendix 10

def mult\_matrix(vector):

  n = len(vector) #this is a crutch to use get\_plot() without any changes

  matrix1 = np.random.rand(n, n) #use numpy for generate random matrixe

  matrix2 = np.random.rand(n, n)

  result = np.dot(matrix1,matrix2) #use numpy to find their product