Title page: Time Complexity Analysis

FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report
on the practical task No. 1
“Time Complexity”

Performed by
Abizer Safdari
J4134c
Accepted by
Dr Petr Chunaev

St. Petersburg
2020

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* $\boldsymbol{v} = [v_1, v_2, \dots, v_n]$ *with non-negative elements. For* $\boldsymbol{v}$*, implement the following calculations and algorithms:*

1) $f(\boldsymbol{v}) = const$ *(constant function);*
2) $f(\boldsymbol{v}) = \sum_{k=1}^{n} v_k$ *(the sum of elements);*
3) $f(\boldsymbol{v}) = \prod_{k=1}^{n} v_k$ *(the product of elements);*
4) *supposing that the elements of* $\boldsymbol{v}$ *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* $P(x) = v_1 + x(v_2 + x(v_3 + \cdots))$*;*
5) *Bubble Sort of the elements of* $\boldsymbol{v}$*;*
6) *Quick Sort of the elements of* $\boldsymbol{v}$*;*
7) *Timsort of the elements of* $\boldsymbol{v}$*.*

II. *Generate random matrices* A *and* B *of size* $n \times n$ *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*

Brief theoretical part

- Time complexity of an algorithm signifies the total time required by the program to run till its completion. The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.
- Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.
- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value. The base value is index 0 and the difference between the two indexes is the offset.
- Bubble sort, sometimes referred to as sinking sort, is a simple 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. Bubblesort uses “Bruteforce” technique. Starting from the first index, compare the first and the second elements.If the first element is greater than the second element, they are swapped.Then compare the second and the third elements. Swap them if they are not in order. The above process goes on until the last element. The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end. In each iteration, the comparison takes place up to the last unsorted element. The array is sorted when all the unsorted elements are placed at their correct positions.
- QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. In quicksort algorithms mostly linked lists are used as data structres, as arrays are merged only to a particular arrays.

- TimSort is a sorting algorithm based on Insertion Sort and Merge Sort.
    - ♦ A stable sorting algorithm works in O(n Log n) time
    - ♦ Used in Java’s Arrays.sort() as well as Python’s sorted() and sort().
    - ♦ First sort small pieces using Insertion Sort, then merges the pieces using merge of merge sort.
- We divide the Array into blocks known as Run. We sort those runs using insertion sort one by one and then merge those runs using combine function used in merge sort. If the size of Array is less than run, then Array get sorted

just by using Insertion Sort. The size of run may vary from 32 to 64 $depending\ upon\ the\ size\ of\ the\ array$

Sr. No	**Algorithm**	**Time complexity**
1	Constant function	O(1)
2	Sum of element of array	O(n)
3	Product of element of array	O(n)
4	Coefficient of polynomial	O(n)
5	Horner’s method	O(n)
6	Bubble Sort	O(n^2)
7	Quick Sort	O(n*log(n))
8	Tim Sort	O(n*log(n))
9	Matrix multiplication	O(n^3)

Results

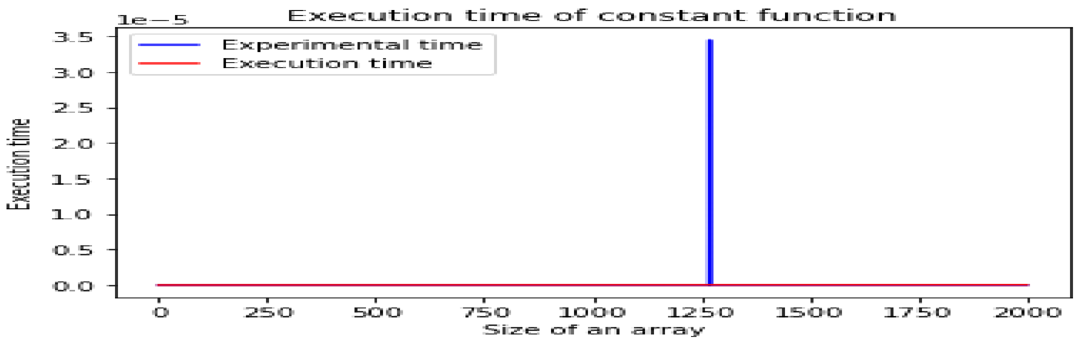

Figure 1: The Experimental time execution for constant function is nearly constant for any size of the array, which matches with the theoretical time complexity O(1) .

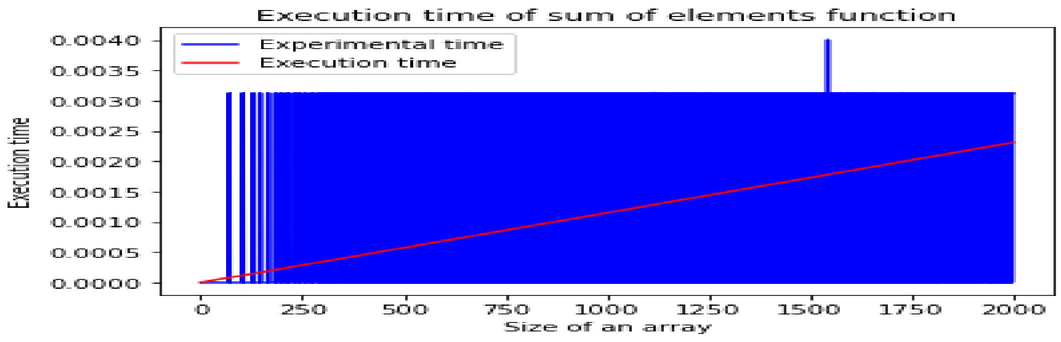

Figure 2: The Experimental time execution for a function, increasing linearly with respect to size of the array, which matches with the theoretical time complexity O(n) .

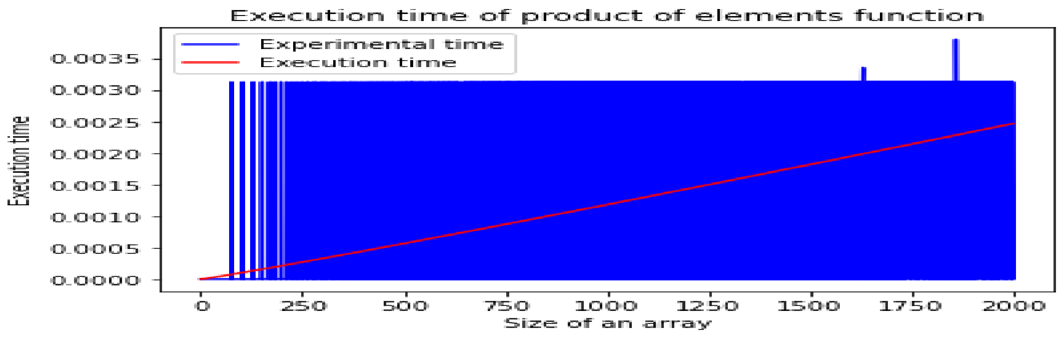

Figure 3: The Experimental time execution for a function, increasing linearly with respect to size of the array, which matches with the theoretical time complexity O(n) .

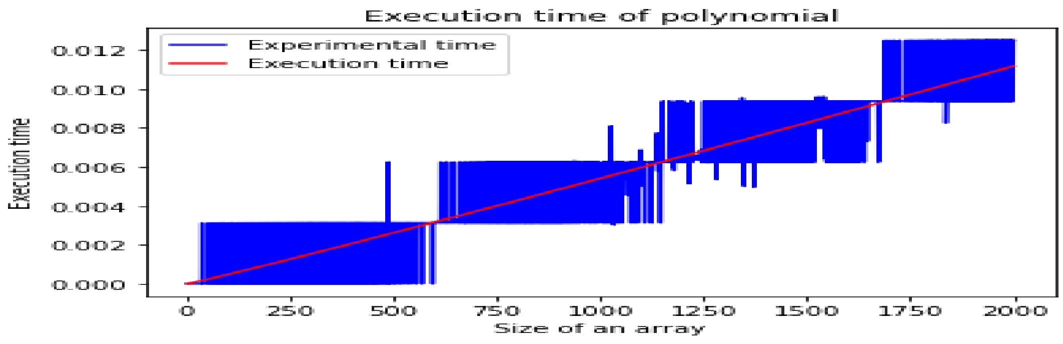

Figure 4: The Experimental time execution for a polynomial function at x=1.5 is increasing linearly with respect to size of the array, which matches with the theoretical time complexity O(n) .

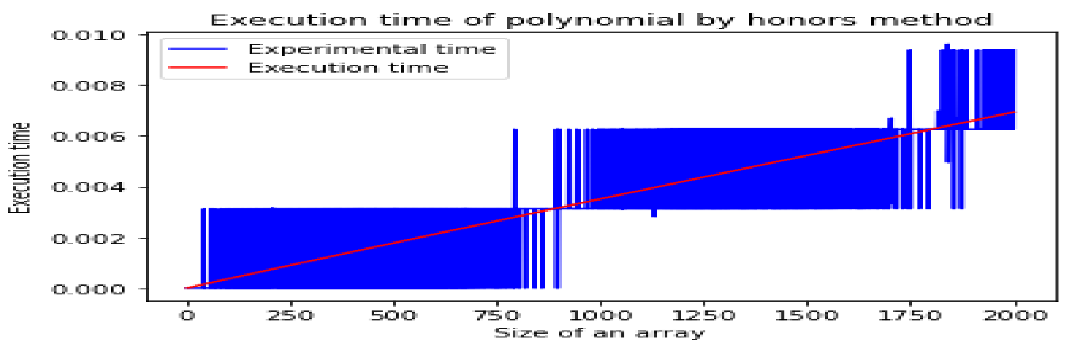

Figure 5: The Experimental time execution for a polynomial function by horner's method at x=1.5 is increasing linearly with respect to size of the array, which matches with the theoretical time complexity O(n) .

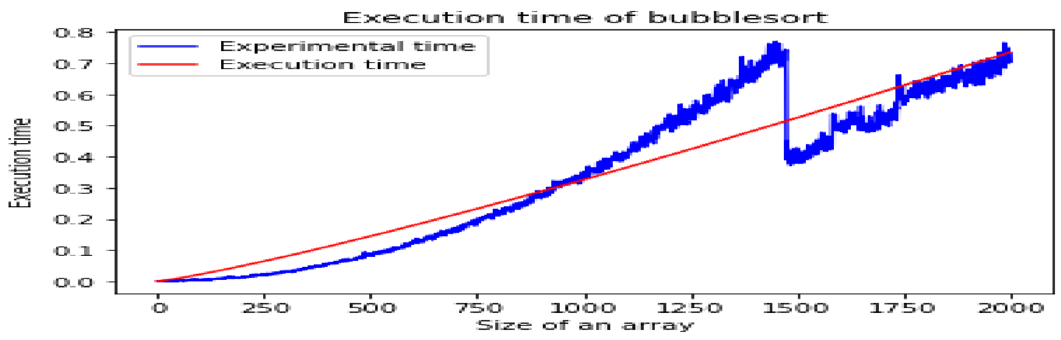

Figure 6: The Experimental time execution for sorting an array using bubble sort algorithm increases with respect to square of size of the array, which matches with the theoretical time complexity. O(n 2) .

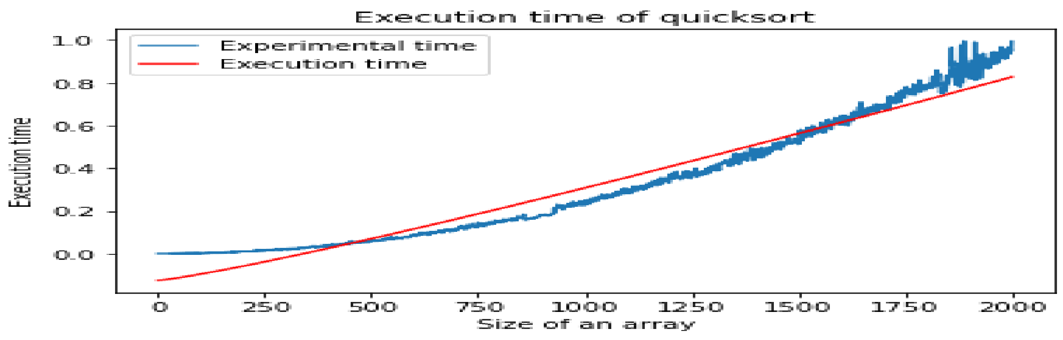

Figure 7: The Experimental time execution for sorting an array using quick sort algorithm increases linearithmicly with respect to of size of the array, which approximately matches with the theoretical time complexity O(n∗log(n)) .

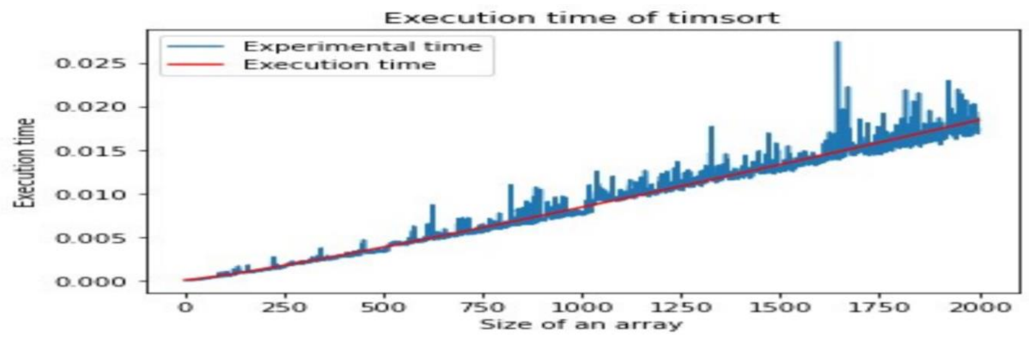

Figure 8: The Experimental time execution for sorting an array using timsort algorithm increases linearithmicly with respect to of size of the array, which approximately matches with the theoretical time complexity O(n∗log(n)) .

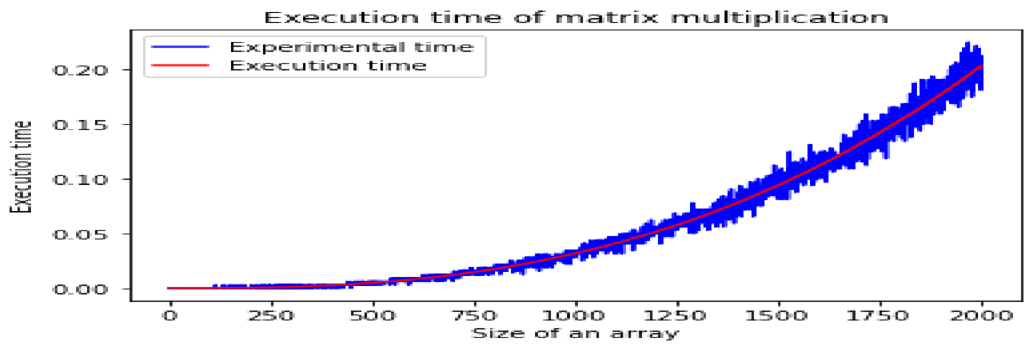

Figure 9: The Experimental time execution for matrix multiplications increases to the power of three with respect to of size of the array, which approximately matches with the theoretical time complexity O(n 3) . Here the time is executed for arrays up to size 100, due to long running time.

Conclusions

- The execution time of constant function does not depends on the size of the array.
- The execution time of functions, which returns product or sum of elements the depends on the size on the array
- Time complexity is similar for solving polynomial by evaluating one by one term and horner’s method. But horner’s method is bit faster due to less elementary operation.
- Execution time of bubble sort increases drastically with increase in size of array. So, bubblesort does not works well with large arrays.
- For sorting an array, quick sort and timsort are preferred due to less execution time.
- The execution time of matrix multiplication is O(n 3) . So, it does not work well with large matrices, due to large execution time.

Appendix

```
"""
Created on Sat Oct  3 10:19:25 2020
@author: abizer
"""
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import time
import numpy as np
from decimal import Decimal
#for constant function
def time1(v):
    a=[]
    for i in range(5):
        start_time=time.time()
        con(v)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def con(v):
    '''returns a condtant value C'''
    return"c"
#for summation in array element
def time2(v):
    a=[]
    n=len(v)
    for i in range(5):
        start_time=time.time()
        soa(v,n)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def soa(v, n):
     if len(v)== 1:
        return v[0]
     else:
        return v[0]+soa(v[1:], n)
#for product of array element
def time3(v):
    a=[]
    n=len(v)
    for i in range(5):
        start_time=time.time()
        poa(v,n)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def poa(v, n):
     if len(v)== 1:
        return v[0]
     else:
        return v[0]+soa(v[1:], n)
#polynomial function
def time4(v):
    a=[]
    for i in range(5):
        start_time=time.time()
        poly(v)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def poly(v):
    '''value of polynomial at x=1.5'''
    k=0
    for i in range(len(v)):
        k+=Decimal(v[i])*(Decimal(1.5)**Decimal(i))
    return k
#honors function
def time5(v):
    a=[]
    for i in range(5):
        start_time=time.time()
        hon(v)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def hon(v):
    '''value of polynomial using horner's at x=1.5'''
    k=0
    for i in range(len(v)):
        k=Decimal(v[i])+Decimal(1.5)*Decimal(k)
    return k
#bubble short
def time6(v):
    a=[]
    for i in range(5):
        start_time=time.time()
        bubbleSort(v)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def bubbleSort(v):
    n=len(v)
    for i in range(n-1):
        for j in range(0,n-1-i):
            if v[j] > v[j+1]:
                v[j], v[j+1] = v[j+1], v[j]
    return v
#for quick sort
def time7(v):
    a=[]
    for i in range(5):
        n = len(v)
        start_time=time.time()
        quickSort(v,0,n-1)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
def partition(arr,low,high):
   i = ( low-1 )
   pivot = arr[high] # pivot element
   for j in range(low , high):
      # If current element is smaller
      if arr[j] <= pivot:
         # increment
         i = i+1
         arr[i],arr[j] = arr[j],arr[i]
   arr[i+1],arr[high] = arr[high],arr[i+1]
   return ( i+1 )
# sort
def quickSort(arr,low,high):
   if low < high:
      # index
      pi = partition(arr,low,high)
      # sort the partitions
      quickSort(arr, low, pi-1)
      quickSort(arr, pi+1, high)
#for timesort
def time8(v):
    a=[]
    for i in range(5):
        start_time=time.time()
        timsort(v)
        a.append(time.time()-start_time)
    a=sum(a)/5
    return a
#defining insertion for using in timsort
def InsertionSort(array):
    '''Sorting the array using Insertion sort'''
    for x in range (1, len(array)):
        for i in range(x, 0, -1):
            if array[i] < array[i - 1]:
                t = array[i]
                array[i] = array[i - 1]
                array[i - 1] = t
            else:
                break
            i = i - 1
    return array
#defining merge for using timsort
def Merge(aArr, bArr):
    '''Merges he given arrays'''
    a = 0
    b = 0
    cArr = []
    while a < len(aArr) and b < len(bArr):
        if aArr[a] <= bArr[b]:
            cArr.append(aArr[a])
            a = a + 1
        elif aArr[a] > bArr[b]:
            cArr.append(bArr[b])
            b = b + 1
    while a < len(aArr):
        cArr.append(aArr[a])
        a = a + 1
    while b < len(bArr):
        cArr.append(bArr[b])
        b = b + 1
    return cArr
#tim sort
def timsort(v):
    '''Sorting the array using Tim sort'''
    for x in range(0, len(v), 64):
        v[x : x + 64] = InsertionSort(v[x : x + 64])
    RUNinc = 64
    while RUNinc < len(v):
        for x in range(0, len(v), 2 * RUNinc):
            v[x : x + 2 * RUNinc] = Merge(v[x : x + RUNinc], v[x + RUNinc: x +
2 * RUNinc])
```

```
    RUNinc = RUNinc * 2
    return v
#for matrix miltiplation
def time9(a,b):
    d=[]
    for j in range(5):
        start_time=time.time()
        r=np.dot(a,b)
        d.append(time.time()-start_time)
    d=sum(d)/5
    return d
#main driver code
n=np.arange(1,2001)
t1=[]
t2=[]
t3=[]
t4=[]
t5=[]
t6=[]
t7=[]
t8=[]
t9=[]
for i in range(1,2001):
    v=np.random.rand(i)
    a=np.array(np.random.rand(i,i))
    b=np.array(np.random.rand(i,i))
    t1.append(time1(v))
    t2.append(time2(v))
    t3.append(time3(v))
    t4.append(time4(v))
    t5.append(time5(v))
    t6.append(time6(v))
    t7.append(time7(v))
    t8.append(time8(v))
    t9.append(time9(a,b))


#Curve Fitting function
def func(x, a, b):
    return a*(x**(b))
# Curve fitting function for linearthamtic time
def func2(x, a, b):
    return a *x* np.log(x ) + b
# Initial guess for the parameters
initialGuess = [0.1,0.1]
```

```
#x values for the fitted function
xFit=np.arange(1,2001)

#Plot experimental data points of execution time of constant function
plt.plot(n, t1, '-b',label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t1, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r', label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of constant function')
plt.show()

#Plot experimental data points of execution time of sum of elements function
plt.plot(n, t2, '-b',label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t2, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r', label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of sum of elements function')
plt.show()

#Plot experimental data points of execution time of product of elements function
plt.plot(n, t3, '-b', label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t3, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r',label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of product of elements function')
plt.show()

#Plot experimental data points of execution time of polynomial
plt.plot(n, t4, '-b', label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t4, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r', label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of polynomial')
plt.show()

#Plot experimental data points of execution time of polynomial by honors method
plt.plot(n, t5, '-b', label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t5, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r', label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of polynomial by honors method')
plt.show()

#Plot experimental data points of execution time of bubble sorting
plt.plot(n, t6, '-b', label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t6, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r', label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of bubblesort')
plt.show()

#Plot experimental data points of execution time of quick sortig
#Perform the curve-fit
popt, pcov = curve_fit(func2, n, t7)
plt.figure()
plt.plot(n, t7, label="Experimental time")
#Plot the fitted function
plt.plot(n, func2(n, *popt), '-r', label="Execution time")
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of quicksort')
plt.show()

#Plot experimental data points of execution time of tim sorting
#Perform the curve-fit
```

```
popt, pcov = curve_fit(func2, n, t8)
plt.figure()
plt.plot(n, t8, label="Experimental time")
#Plot the fitted function
plt.plot(n, func2(n, *popt), '-r', label="Execution time")
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of timsort')
plt.show()

#Plot experimental data points for execution time of matrix multiplication
plt.plot(n, t9, '-b', label='Experimental time')
#Perform the curve-fit
popt, pcov = curve_fit(func, n, t9, initialGuess)
#Plot the fitted function
plt.plot(xFit, func(xFit, *popt), '-r', label='Execution time')
plt.legend()
plt.xlabel('Size of an array')
plt.ylabel('Execution time')
plt.title('Execution time of matrix multiplication')
plt.show()
```