

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)
Кафедра комплексной информационной безопасности электронно-
вычислительных систем (КИБЭВС)

ОЦЕНКА АЛГОРИТМИЧЕСКОЙ СЛОЖНОСТИ

Отчет по лабораторной работе №1

По дисциплине «Технологии и методы программирования»

Выполнил

Студент гр. 728-1

_____ Медведев З.М.

_____ 2020

Принял

Преподаватель кафедры КИБЭВС

_____ Перминов П.В.

_____ 2020

Оглавление

1 Введение.....	3
2 Ход работы.....	4
3 Заключение	19
Приложение А (Обязательное) Листинг программы для генерации списков	20
Приложение Б (Обязательное) Листинг программы сортировка расческой	21
Приложение В (Обязательное) Листинг программы сортировки Шелла	23
Приложение Г (Обязательное) Листинг программы быстрой сортировки.....	26
Приложение Д (Обязательное) Листинг программы пирамидальной сортировки...	28

1 Введение

Целью данной лабораторной работы является оценка алгоритмической сложности алгоритмов сортировки: “Расческа”, Шелла, Быстрая сортировка, Пирамидальная сортировка.

2 Ход работы

Первым делом были сгенерированы массивы(списков), с длинами равными: 1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 50, 75, 100, 250, 500. Для каждой длины были сгенерированы тысяча массивов (списков). Для этого использовался метод GenerateList (Приложение А). Пример массивов(списков) длиной три приведен ниже (рисунок 2.1)

```

, 1900766807], [832256419, 1961372720, 819993150], [1365306743, 1180101026, 992289160], [396315907, 1706082107, 14000367
61], [884205742, 441873225, 1992523702], [1155386400, 1883229169, 150404078], [1922566571, 69590435, 502832415], [208561
9075, 1548660909, 850870495], [1990000934, 1339594109, 1469298953], [1688353180, 245246144, 1345609875], [901182690, 191
0693927, 1386933804], [1842955593, 986581721, 806648325], [834605420, 967814420, 1142965478], [215612523, 121603936, 149
6794028], [14643094, 441888361, 371293153], [1461424164, 1111802643, 1434027720], [600677770, 1211354725, 605739043], [8
45100285, 14661605, 692735579], [471289335, 1587187763, 1033298269], [1137607554, 1686500094, 401653012], [531904249, 18
34891565, 1507949924], [675560591, 592588793, 1863995695], [1851492273, 587802640, 66083262], [997679141, 174073977, 124
5571124], [255357113, 134259337, 2087927919], [406803768, 1983867716, 1643107602], [450310629, 28230491, 1507932158], [3
25292125, 1534971471, 752339204], [1180662273, 634636997, 619525957], [269092618, 1244871697, 1576668738], [70401651, 67
5004180, 1208551958], [638094019, 1411283690, 572998575], [114330335, 1878226116, 952403155], [587874894, 2044055502, 29
8702287], [1102520252, 1649681772, 1086998345], [780306112, 1804082481, 1138833907], [1176364671, 562340186, 1452470332],
, [1774069301, 1147209606, 241150488], [542198494, 712981106, 1270762851], [769443416, 850692186, 32064625], [1545136252
, 1622584466, 1859606013], [1667826586, 44560734, 2068541456], [1746360460, 1525260659, 1845759696], [550894336, 9212150
76, 1782979011], [1937287013, 820345504, 633238280], [1097190263, 1819594198, 1772832820], [1704991156, 1518431814, 1940
942355], [279063231, 78437237, 559605653], [782663700, 725099159, 917993984], [2015967408, 504451859, 848838917], [15832
77430, 1404740417, 538172902], [1009739123, 1489192071, 1307906761], [1816811136, 18849387, 1895957318], [1205846125, 82
0338675, 1652545523], [2026398021, 704623147, 123608174], [844100164, 1998781127, 2115844520], [1970134928, 1030896920,
752375041], [2128515858, 1138354582, 666300271], [1278710655, 51398869, 1803273464], [1365646572, 2021696823, 1132230177
], [2032902612, 74584330, 317536080], [1852507807, 777360878, 600228144], [262797342, 1591051049, 157142148], [252291256
, 729849392, 320043268], [984801282, 779677987, 127004164], [1443920394, 856554744, 544269641], [1188524118, 824874172,
1844000801], [752493590, 2049520786, 1626682028], [580042614, 699977726, 1305388400], [386167912, 761637146, 1216775954],
, [1375000598, 871373042, 485505060], [1105924994, 2100328061, 1404063777], [1174655056, 718976818, 1396377499], [691944
093, 965944670, 1074294794], [567962758, 337561497, 1812156797], [1738623407, 1487983329, 2126698802], [1205765274, 2972
52289, 872651990], [452900990, 275581719, 634800115], [309221690, 27351593, 1958598356], [1519633648, 1860928203, 609248
279], [1700528265, 45918698, 19981626], [1398331906, 889006586, 240190284], [1673428731, 1388376936, 970790638], [126151
8180, 1266740555, 473806541], [2021933756, 866210960, 2029775570], [1760381839, 646281722, 1574682265], [965533282, 2109
549019, 1469684871], [1906872217, 871452866, 801607164], [1733948787, 1558114663, 1221388797], [711026008, 1551509677, 3
45959625], [2020513894, 422544513, 195537183]]

```

Рисунок 2.1 – Сгенерированные массивы(списки) длиной три

Затем группы массивов(списков) каждой длины были отсортированы всеми четырьмя алгоритмами поочередно, также было подсчитано количество операций и время затраченное на сортировку.

Первым алгоритмом, который использовался для операций, перечисленных выше, является сортировка расческой (рисунок 2.2) и (рисунок 2.3). Исходный код программы, написанной на языке Python, находится ниже (Приложение Б)

```

===== Сортировка расческой =====
Длина списка: 1      Количество операций: 0

real    0m0.039s
user    0m0.034s
sys     0m0.003s
Длина списка: 2      Количество операций: 1464

real    0m0.054s
user    0m0.047s
sys     0m0.007s
Длина списка: 3      Количество операций: 4308

real    0m0.043s
user    0m0.032s
sys     0m0.010s
Длина списка: 4      Количество операций: 8265

real    0m0.045s
user    0m0.041s
sys     0m0.004s
Длина списка: 5      Количество операций: 12492

real    0m0.059s
user    0m0.054s
sys     0m0.004s
Длина списка: 10     Количество операций: 42161

real    0m0.063s
user    0m0.049s
sys     0m0.013s
Длина списка: 15     Количество операций: 84168

real    0m0.090s
user    0m0.064s
sys     0m0.022s
Длина списка: 20     Количество операций: 129532

real    0m0.105s
user    0m0.101s
sys     0m0.004s
Длина списка: 25     Количество операций: 176680

real    0m0.124s
user    0m0.117s
sys     0m0.007s

```

Рисунок 2.2 – Количество операций и затраченное время на сортировку Расческой

```

Длина списка: 30     Количество операций: 235313

real    0m0.151s
user    0m0.135s
sys     0m0.013s
Длина списка: 50     Количество операций: 471357

real    0m0.245s
user    0m0.240s
sys     0m0.004s
Длина списка: 75     Количество операций: 844540

real    0m0.379s
user    0m0.377s
sys     0m0.000s
Длина списка: 100    Количество операций: 1238323

real    0m0.523s
user    0m0.515s
sys     0m0.007s
Длина списка: 250    Количество операций: 4110862

real    0m1.541s
user    0m1.527s
sys     0m0.010s
Длина списка: 500    Количество операций: 9361517

real    0m3.542s
user    0m3.511s
sys     0m0.023s

```

Рисунок 2.3 – Количество операций и затраченное время на сортировку Расческой

Далее был задействован алгоритм сортировки Шелла (рисунок 2.4) и (рисунок 2.5). Реализация алгоритма сортировки Шелла на языке Python находится ниже (Приложение В).

```

===== Сортировка Шелла =====
Длина списка: 1      Количество операций: 0

real    0m0.033s
user    0m0.029s
sys     0m0.004s
Длина списка: 2      Количество операций: 1504

real    0m0.036s
user    0m0.032s
sys     0m0.004s
Длина списка: 3      Количество операций: 3512

real    0m0.041s
user    0m0.030s
sys     0m0.010s
Длина списка: 4      Количество операций: 7322

real    0m0.042s
user    0m0.038s
sys     0m0.004s
Длина списка: 5      Количество операций: 10574

real    0m0.046s
user    0m0.045s
sys     0m0.000s
Длина списка: 10     Количество операций: 34047

real    0m0.056s
user    0m0.055s
sys     0m0.000s
Длина списка: 15     Количество операций: 57623

real    0m0.069s
user    0m0.068s
sys     0m0.000s
Длина списка: 20     Количество операций: 98051

real    0m0.088s
user    0m0.085s
sys     0m0.004s
Длина списка: 25     Количество операций: 136663

real    0m0.110s
user    0m0.099s
sys     0m0.010s
Длина списка: 30     Количество операций: 160708

```

Рисунок 2.4 – Количество операций и затраченное время на сортировку Шеллом

```

Длина списка: 50     Количество операций: 361765

real    0m0.221s
user    0m0.220s
sys     0m0.000s
Длина списка: 75     Количество операций: 616486

real    0m0.331s
user    0m0.323s
sys     0m0.007s
Длина списка: 100    Количество операций: 917511

real    0m0.491s
user    0m0.483s
sys     0m0.007s
Длина списка: 250    Количество операций: 2736402

real    0m1.248s
user    0m1.235s
sys     0m0.010s
Длина списка: 500    Количество операций: 6586160

real    0m3.005s
user    0m2.979s
sys     0m0.020s

```

Рисунок 2.5 – Количество операций и затраченное время на сортировку Шеллом

Третий алгоритм, который был задействован, оказался алгоритмически самым выгодным, он имеет название Быстрая сортировка (рисунок 2.6, рисунок 2.7). Исходный код программы приведен в Приложение Г.

```

===== Быстрая сортировка =====
Длина списка: 1          Количество операций: 500500

real    0m0.037s
user    0m0.033s
sys     0m0.003s
Длина списка: 2          Количество операций: 1501500

real    0m0.040s
user    0m0.037s
sys     0m0.003s
Длина списка: 3          Количество операций: 2142498

real    0m0.042s
user    0m0.039s
sys     0m0.003s
Длина списка: 4          Количество операций: 2838508

real    0m0.038s
user    0m0.038s
sys     0m0.000s
Длина списка: 5          Количество операций: 3518988

real    0m0.049s
user    0m0.039s
sys     0m0.010s
Длина списка: 10         Количество операций: 6788014

real    0m0.061s
user    0m0.057s
sys     0m0.004s
Длина списка: 15         Количество операций: 10210778

real    0m0.074s
user    0m0.070s
sys     0m0.004s
Длина списка: 20         Количество операций: 13475718

real    0m0.086s
user    0m0.075s
sys     0m0.010s
Длина списка: 25         Количество операций: 16832630

```

Рисунок 2.6 – Работа алгоритма Быстрой сортировки

```

Длина списка: 30      Количество операций: 20231948
real    0m0.112s
user    0m0.095s
sys     0m0.017s
Длина списка: 50      Количество операций: 33548256
real    0m0.172s
user    0m0.162s
sys     0m0.007s
Длина списка: 75      Количество операций: 50333392
real    0m0.261s
user    0m0.256s
sys     0m0.004s
Длина списка: 100     Количество операций: 66847260
real    0m0.335s
user    0m0.329s
sys     0m0.004s
Длина списка: 250     Количество операций: 167162086
real    0m0.847s
user    0m0.832s
sys     0m0.010s
Длина списка: 500     Количество операций: 333571466
real    0m1.678s
user    0m1.652s
sys     0m0.017s

```

Рисунок 2.7 – Работа алгоритма Быстрой сортировки

Еще одна сортировка носит название пирамидальная, количество операций и затраченное время при сортировке приведено ниже (рисунок 2.8) и (рисунок 2.9). Исходный код реализации алгоритма приведен в Приложении Д.

```

===== Пирамидальная сортировка =====
Длина списка: 1      Количество операций: 0
real    0m0.036s
user    0m0.029s
sys     0m0.007s
Длина списка: 2      Количество операций: 3498
real    0m0.038s
user    0m0.038s
sys     0m0.000s
Длина списка: 3      Количество операций: 6706
real    0m0.040s
user    0m0.039s
sys     0m0.000s
Длина списка: 4      Количество операций: 11967
real    0m0.044s
user    0m0.040s
sys     0m0.004s
Длина списка: 5      Количество операций: 16358
real    0m0.047s
user    0m0.043s
sys     0m0.004s
Длина списка: 10     Количество операций: 47742
real    0m0.072s
user    0m0.065s
sys     0m0.007s
Длина списка: 15     Количество операций: 83751

```

Рисунок 2.8 – Работа алгоритма Пирамидальной сортировки


```

Длина списка: 20      Количество операций: 124058
real    0m0.122s
user    0m0.114s
sys     0m0.007s
Длина списка: 25      Количество операций: 166407
real    0m0.152s
user    0m0.148s
sys     0m0.004s
Длина списка: 30      Количество операций: 212386
real    0m0.183s
user    0m0.176s
sys     0m0.007s
Длина списка: 50      Количество операций: 407853
real    0m0.304s
user    0m0.299s
sys     0m0.004s
Длина списка: 75      Количество операций: 676152
real    0m0.505s
user    0m0.500s
sys     0m0.004s
Длина списка: 100     Количество операций: 964856
real    0m0.670s
user    0m0.648s
sys     0m0.020s
Длина списка: 250     Количество операций: 2913981
real    0m2.002s
user    0m1.989s
sys     0m0.007s
Длина списка: 500     Количество операций: 6578960
real    0m4.419s
user    0m4.396s
sys     0m0.010s

```

Рисунок 2.9 – Работа алгоритма Пирамидальной сортировки

Далее было вычислено среднее время и среднее количество операций, для всех четырех алгоритмов, на всех пятнадцати наборах данных (рисунок 2.10).

```

Average counts for comb: 1115514
Average counts for shell: 782012
Average counts for fast: 48624903
Average counts for heap: 814198
Average time for comb: 0.4321333333333337
Average time for shell: 0.36606666666666665
Average time for fast: 0.25566666666666665
Average time for heap: 0.5618

```

Рисунок 2.10 – Среднее время и среднее количество операций для всех алгоритмов

Были построены графики зависимости количества операций от длины массива и графики зависимости времени от длины массива. Также были добавлены графики зависимостей. Сортировка расческой (рисунок 2.11) и (рисунок 2.12). Сортировка Шелла (рисунок 2.13) и (рисунок 2.14). Быстрая сортировка (рисунок 2.15) и (рисунок 2.16). Пирамидальная сортировка (рисунок 2.17) и (рисунок 2.18).

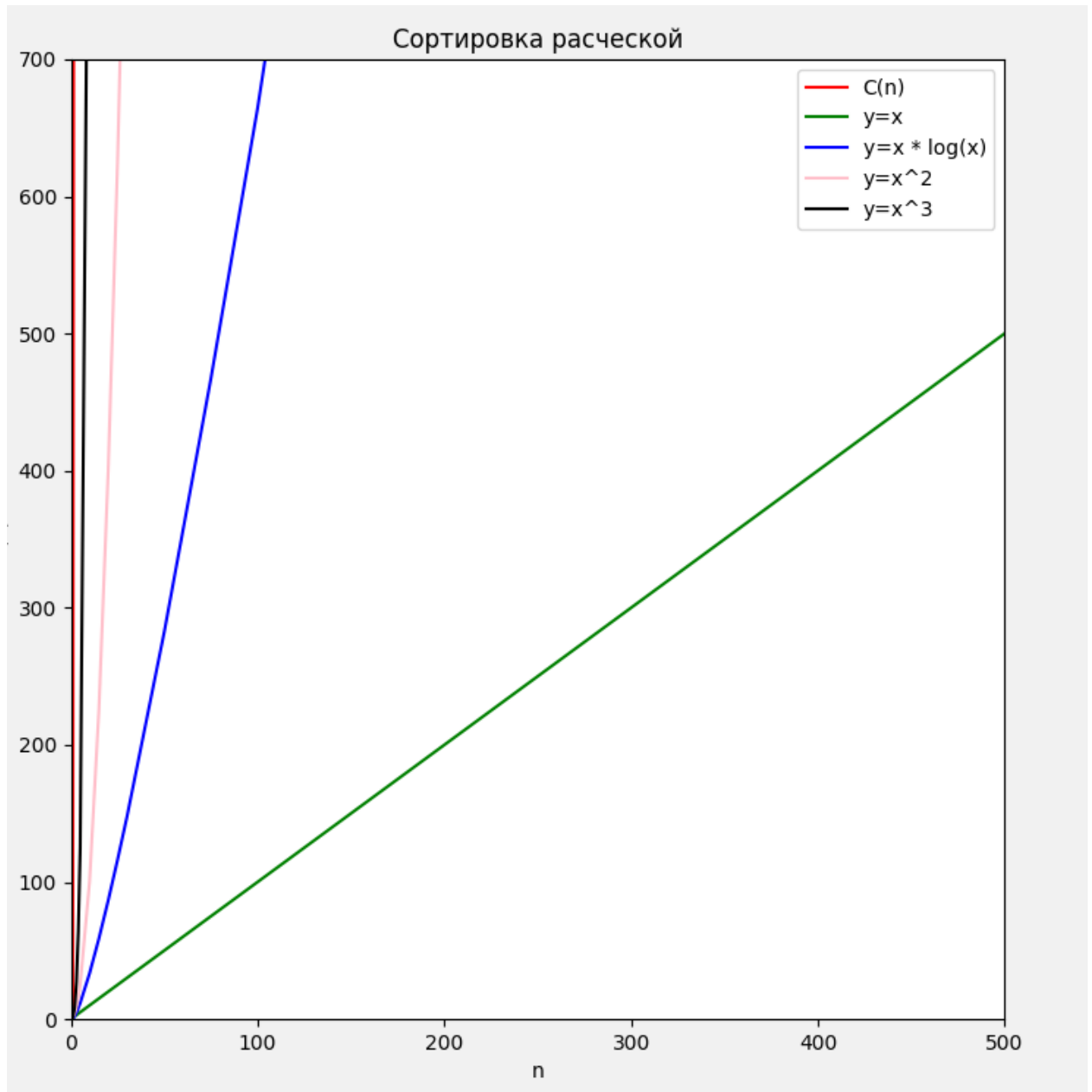


Рисунок 2.11 – График зависимости количества операций от длины массива для сортировки Расческой

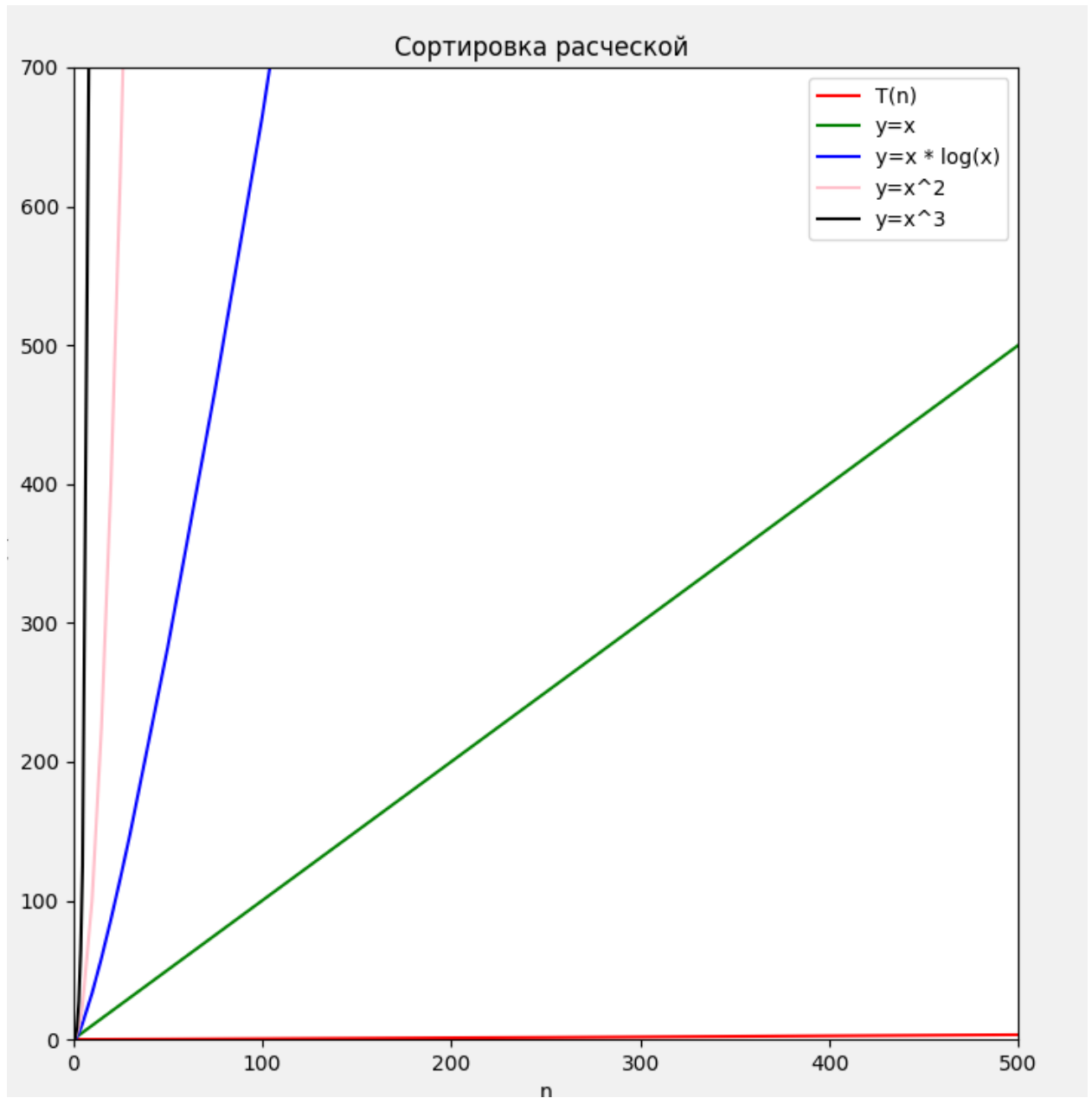


Рисунок 2.12 – График зависимости времени от длины массива для сортировки Расческой

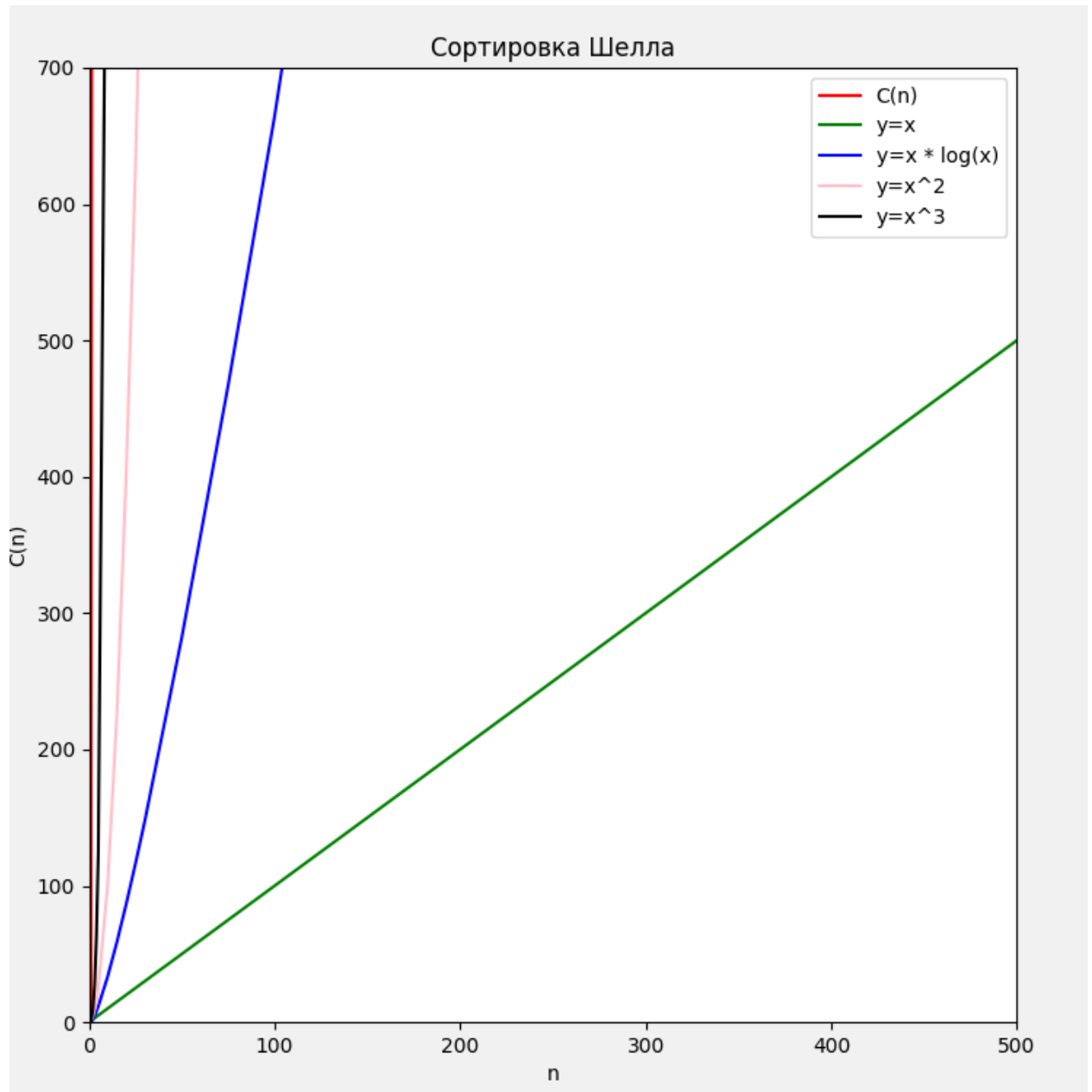


Рисунок 2.13 – График зависимости количества операций от длины массива для сортировки Шелла

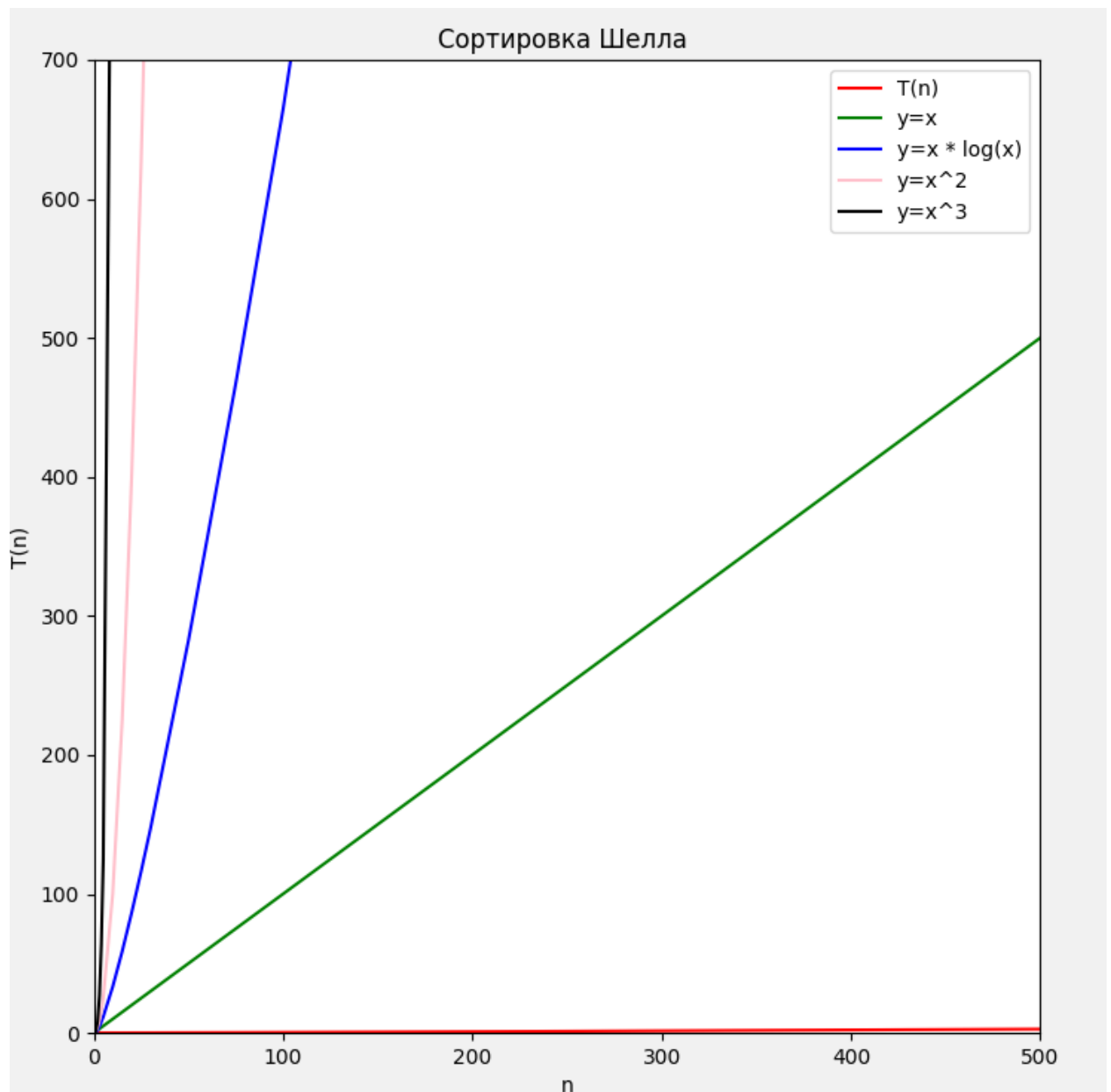


Рисунок 2.14 – График зависимости времени от длины массива для сортировки Шелла

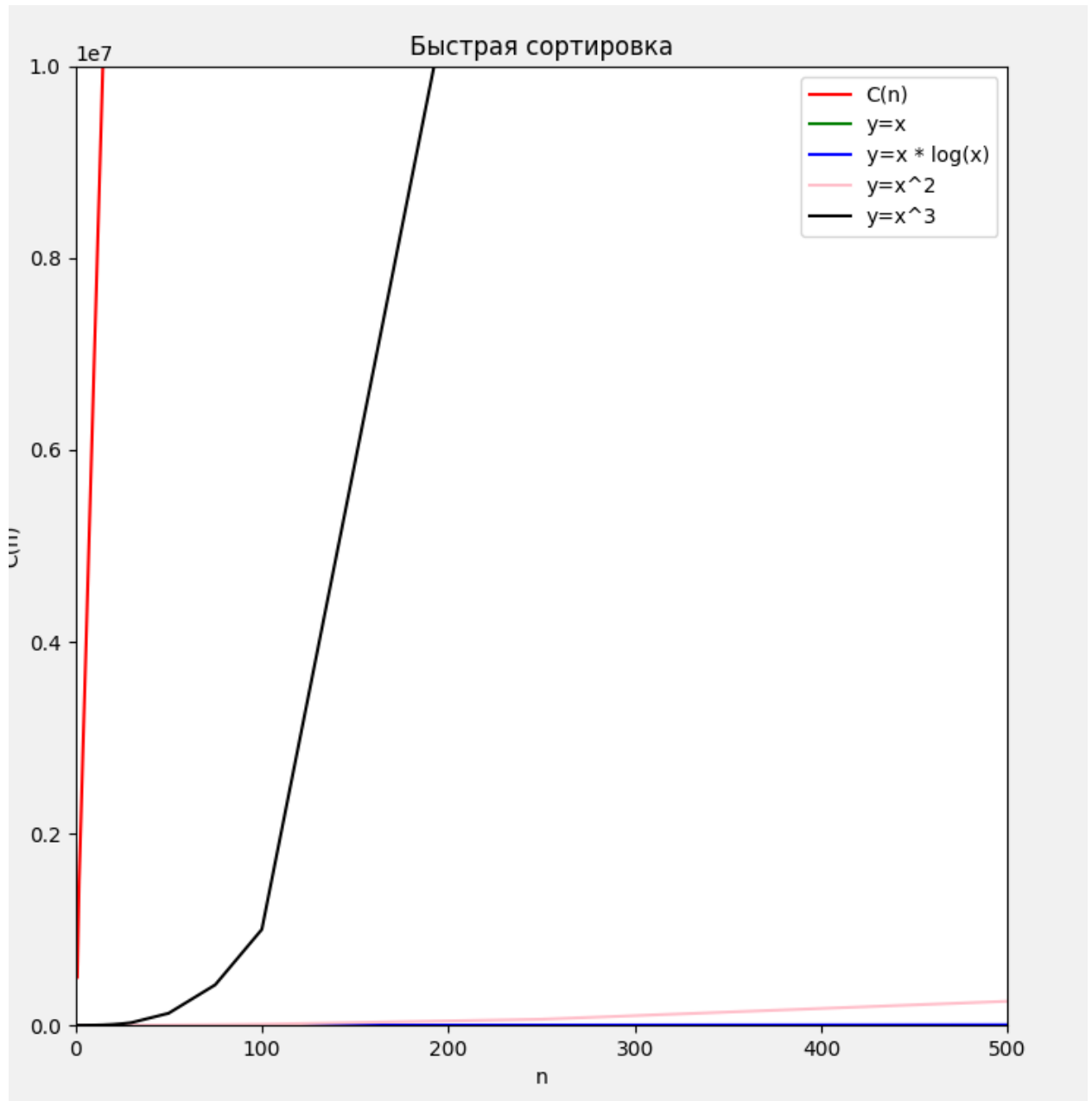


Рисунок 2.15 – График зависимости количества операций от длины массива для Быстрой сортировки

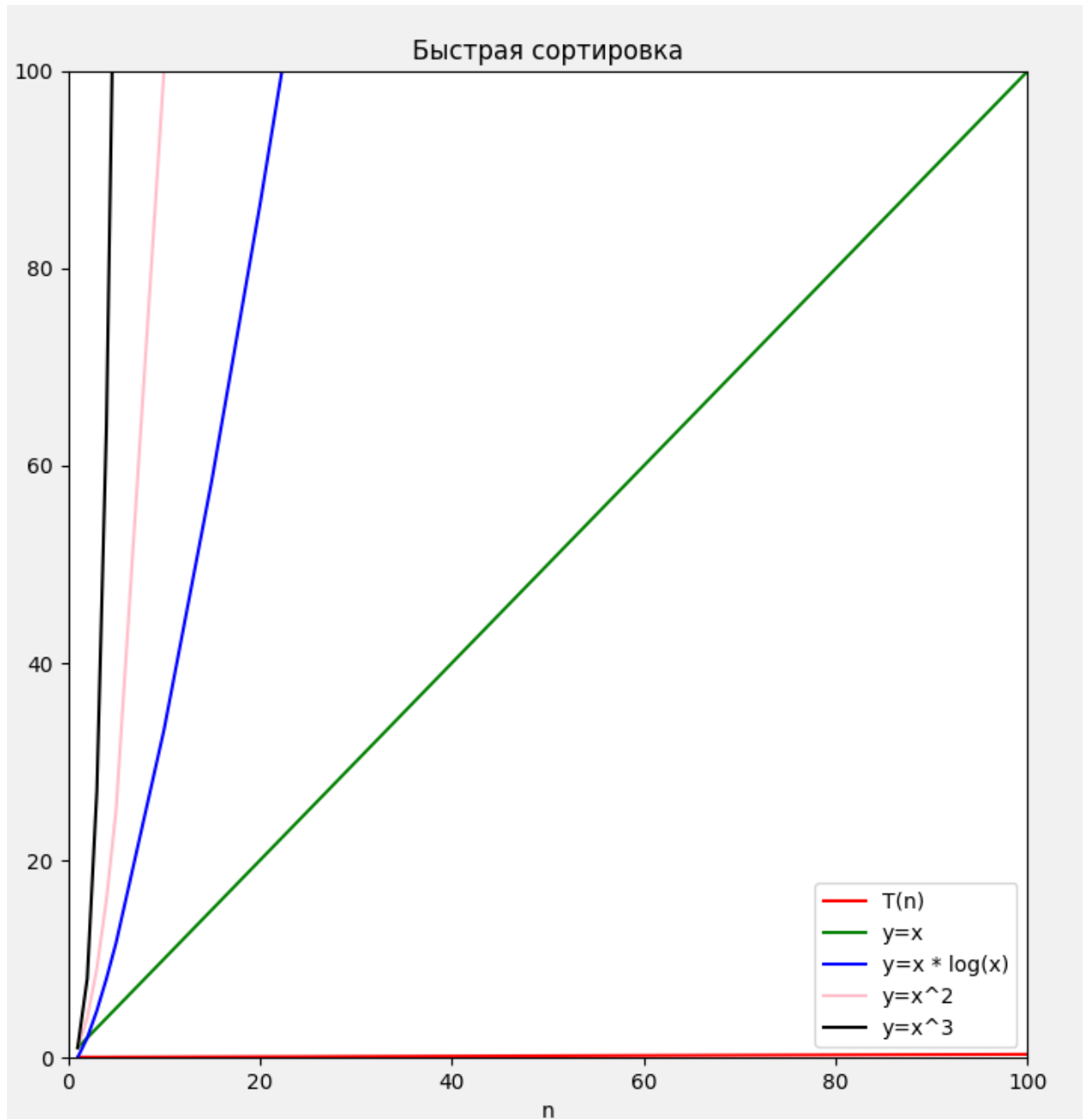


Рисунок 2.16 – График зависимости времени от длины массива для Быстрой сортировки

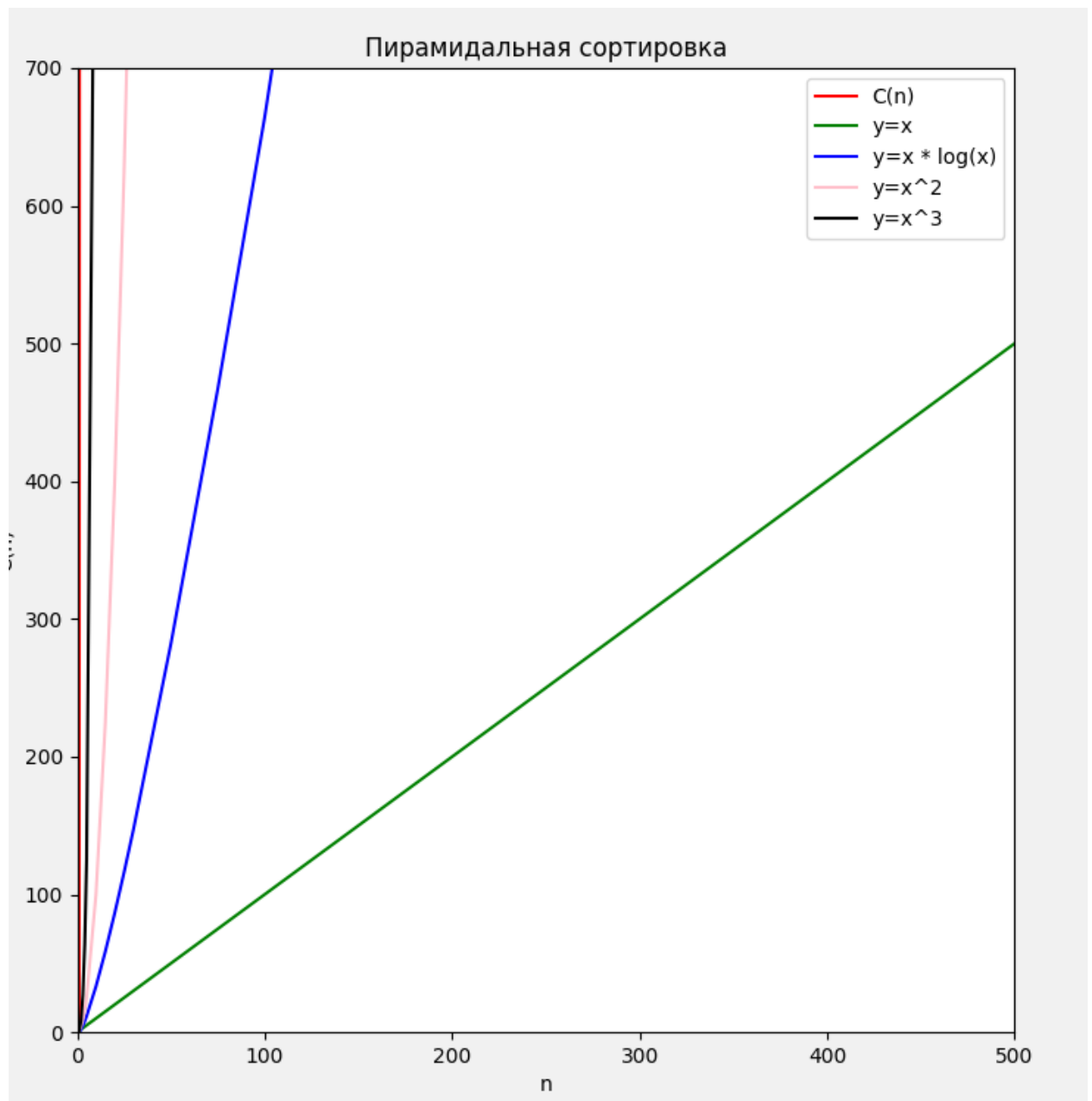


Рисунок 2.17 – График зависимости количества операций от длины массива для Пирамидальной сортировки

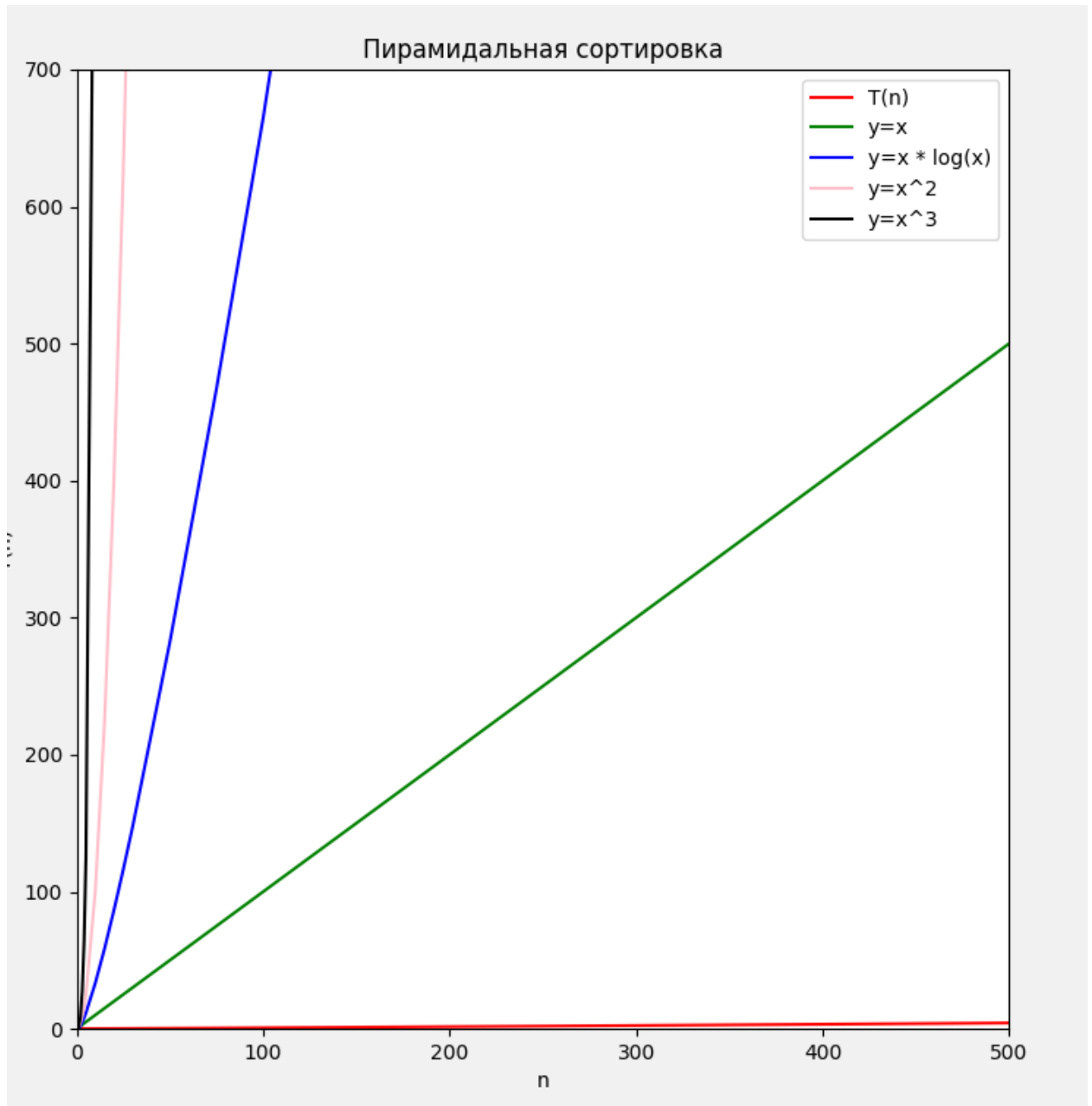


Рисунок 2.18 – График зависимости времени от длины массива для Пирамидальной сортировки

Таким образом из всех сортировок самой алгоритмически выгодной оказалась Быстрая сортировка, её алгоритмическая сложность представлена функцией ниже (формула 2.1)

$$O(n * \log(n)) \quad (2.1)$$

Также было получено худшее и лучшее время для массивов (списков) длиной сто элементов при ста запусках программы для всех алгоритмов (рисунок 2.19)

```
===== Сортировка расческой =====  
Лучшее время: 0.472  
Худшее время: 0.524  
  
===== Сортировка Шелла=====  
Лучшее время: 0.415  
Худшее время: 0.457  
  
===== Быстрая сортировка =====  
Лучшее время: 0.283  
Худшее время: 0.34  
  
===== Пирамидальная сортировка =====  
Лучшее время: 0.598  
Худшее время: 1.042
```

Рисунок 2.19 – Лучшее и худшее время работы каждого алгоритма

3 Заключение

В ходе выполнения данной лабораторной работы была проведена оценка алгоритмической сложности алгоритмов сортировки: “Расческа”, Шелла, Быстрая сортировка, Пирамидальная сортировка.

Был написан отчет согласно ОС ТУСУР 01-2013

Приложение А

(Обязательное)

Листинг программы для генерации списков

```
class GenerateList(object):

    def __init__(self, length):

        self.counts = 1000

        self.length = length

    def get_single_list(self, length):

        return [randint(0, sys.maxsize) for _ in range(self.length)] # method for generate
list

    def gen_lists(self):

        return [self.get_single_list(self.length) for _ in range(self.counts)] # method for
generate lists
```

Приложение Б

(Обязательное)

Листинг программы сортировка расческой

```
import sys

from main import *

class CombSort(object):

    def __init__(self, lst):

        self._count = 0

        self._lst = lst

    def __str__(self):

        return "===== Сортировка расчёской ====="

    def comb_sort(self):

        step = len(self._lst)

        swapped = True

        while step > 1 or swapped:

            step = max(1, int((step * 10) / 13))

            swapped = False

            for i in range(len(self._lst) - step):

                self._count += 1

                if self._lst[i] > self._lst[i + step]:

                    self._lst[i], self._lst[i + step] = self._lst[i + step], self._lst[i]
```

```
swapped = True
```

```
def get_result(self):
    self.comb_sort()
    return [self._lst, self._count]
```

```
def main():
    for length in list(map(int, sys.argv[1:])): # цикл для всех
        tmp_count = 0
        for current_list in GenerateList(length).gen_lists(): # цикл для получения одного
            tmp_obj = CombSort(current_list)
            tmp_count += tmp_obj.get_result()[1]
        print(f"Длина списка: {length} \tКоличество операций: {tmp_count}")

if __name__ == '__main__':
    main()
```

Приложение В

(Обязательное)

Листинг программы сортировки Шелла

```
import sys

from main import *


class ShellSort(object):

    def __init__(self, lst):

        self._count = 0

        self.lst = lst

    def __str__(self):

        return "===== Сортировка Шелла ====="

    def shell_sort(self):

        step = len(self.lst) // 2 # шаг

        while step > 0:

            for i in range(step, len(self.lst)):
```

```
self._count += 1

tmp = self.lst[i]

j = i

while j >= step and self.lst[j - step] > tmp:

    self._count += 1

    self.lst[j] = self.lst[j - step]

    j = j - step

self.lst[j] = tmp

step //= 2

return self.lst


def get_result(self):

    self.shell_sort()

    return [self.lst, self._count]


def main():

    for length in list(map(int, sys.argv[1:])): # цикл для всех
```



```
tmp_count = 0

for current_list in GenerateList(length).gen_lists(): # цикл для получения одного
из тысячи списков

    tmp_obj = ShellSort(current_list)

    tmp_count += tmp_obj.get_result()[1]

print(f'Длина списка: {length} \tКоличество операций: {tmp_count}')
```

Приложение Г

(Обязательное)

Листинг программы быстрой сортировки

```
import sys

from main import *

count = 0

def fast_sort(lst):

    global count

    count += 1

    if len(lst) < 2:

        return lst

    else:

        pivot = lst[0] # опорная точка

        min_arr = [i for i in lst[1:] if i < pivot] # список с элементами меньше опорного
элемент.

        max_arr = [i for i in lst[1:] if i >= pivot] # список с элементами больше опорного
элемент.

        return fast_sort(min_arr) + [pivot] + fast_sort(max_arr)
```

```
def get_fast_sort(lst):
```

```
    return [fast_sort(lst), count]
```

```
def main():
```

```
    for length in list(map(int, sys.argv[1:])): # цикл для всех
```

```
        tmp_count = 0
```

```
        average_count = 0
```

```
        for current_list in GenerateList(length).gen_lists(): # цикл для получения одного  
из тысячи списков
```

```
            tmp_obj = get_fast_sort(current_list)
```

```
            tmp_count += tmp_obj[1]
```

```
    print(f"Длина списка: {length} \tКоличество операций: {tmp_count}")
```

Приложение Д

(Обязательное)

Листинг программы пирамидальной сортировки

```
from main import *
```

```
import sys
```

```
class HeapSort(object):
```

```
    def __init__(self, lst):
```

```
        self._lst = lst
```

```
        self._count = 0
```

```
    def __str__(self):
```

```
        return "Пирамидальная сортировка"
```

```
    def heapify(self):
```

```
        start = (len(self._lst) - 2) // 2
```

```
        while start >= 0:
```

```
            self._count += 1
```

```
self.sift_down(start, len(self._lst) - 1)
```

```
start -= 1
```

```
def sift_down(self, start, end):
```

```
    root = start
```

```
    while root * 2 + 1 <= end:
```

```
        child = root * 2 + 1
```

```
        if child + 1 <= end and self._lst[child] < self._lst[child + 1]:
```

```
            self._count += 1
```

```
            child += 1
```

```
        if child <= end and self._lst[root] < self._lst[child]:
```

```
            self._count += 1
```

```
            self._lst[root], self._lst[child] = self._lst[child], self._lst[root]
```

```
            root = child
```

```
        else:
```

```
            return
```

```
def run(self):
```

```
self.heapify()
```

```
end = len(self._lst) - 1
```

```
while end > 0:
```

```
    self._lst[end], self._lst[0] = self._lst[0], self._lst[end]
```

```
    self._count += 1
```

```
    self.sift_down(0, end - 1)
```

```
    self._count += 1
```

```
    end -= 1
```

```
def get_result(self):
```

```
    self.run()
```

```
    return [self._lst, self._count]
```

```
def main():
```

```
    for length in list(map(int, sys.argv[1:])): # ЦИКЛ ДЛЯ ВСЕХ
```

```
        tmp_count = 0
```

```
for current_list in GenerateList(length).gen_lists(): # цикл для получения одного  
из тысячи списков
```

```
    tmp_obj = HeapSort(current_list)
```

```
    tmp_count += tmp_obj.get_result()[1]
```

```
print(f'Длина списка: {length} \tКоличество операций: {tmp_count}')
```

```
if __name__ == '__main__':
```

```
    main()
```