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

Томск 2020

Оглавление

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)

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)

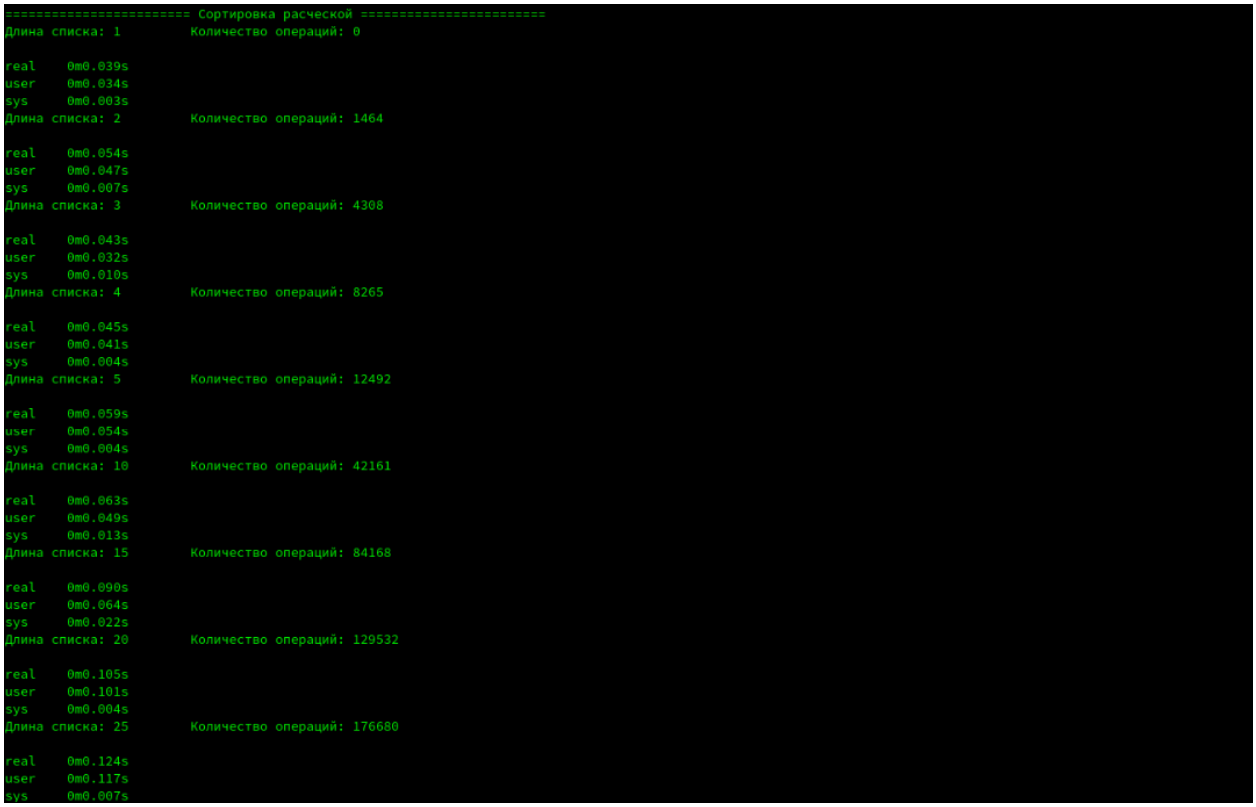

Рисунок 2.2 – Количество операций и затраченное время на сортировку Расческой

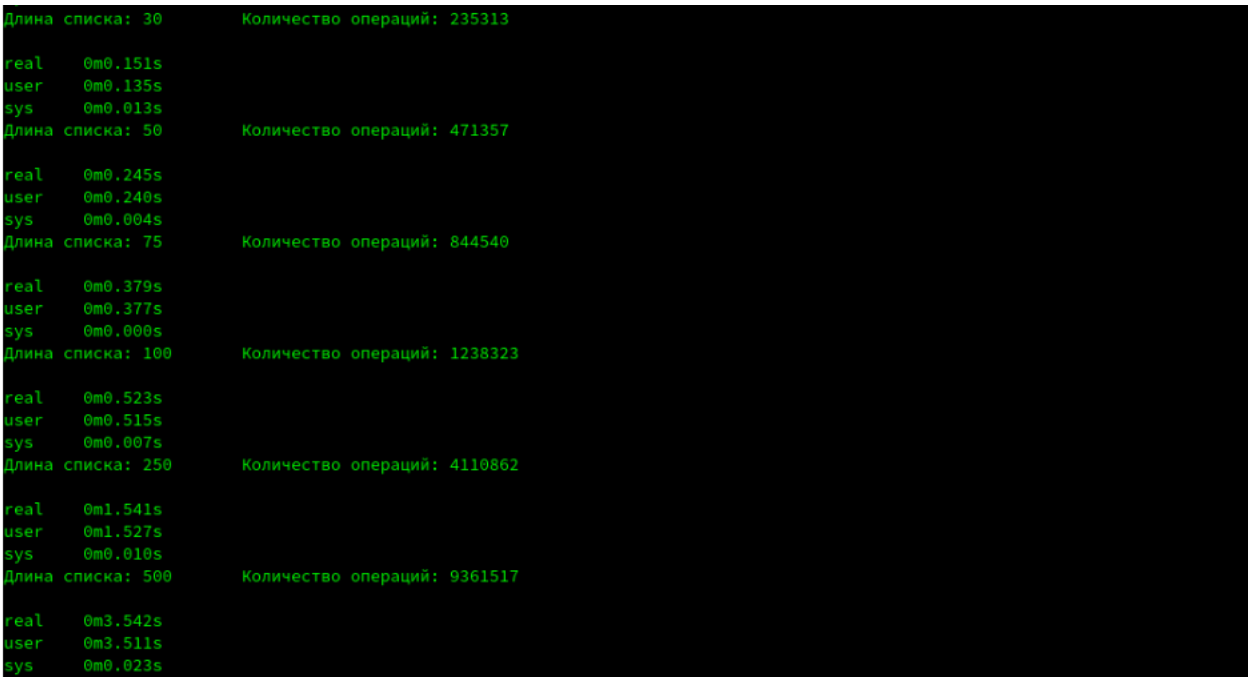

Рисунок 2.3 – Количество операций и затраченное время на сортировку Расческой

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.

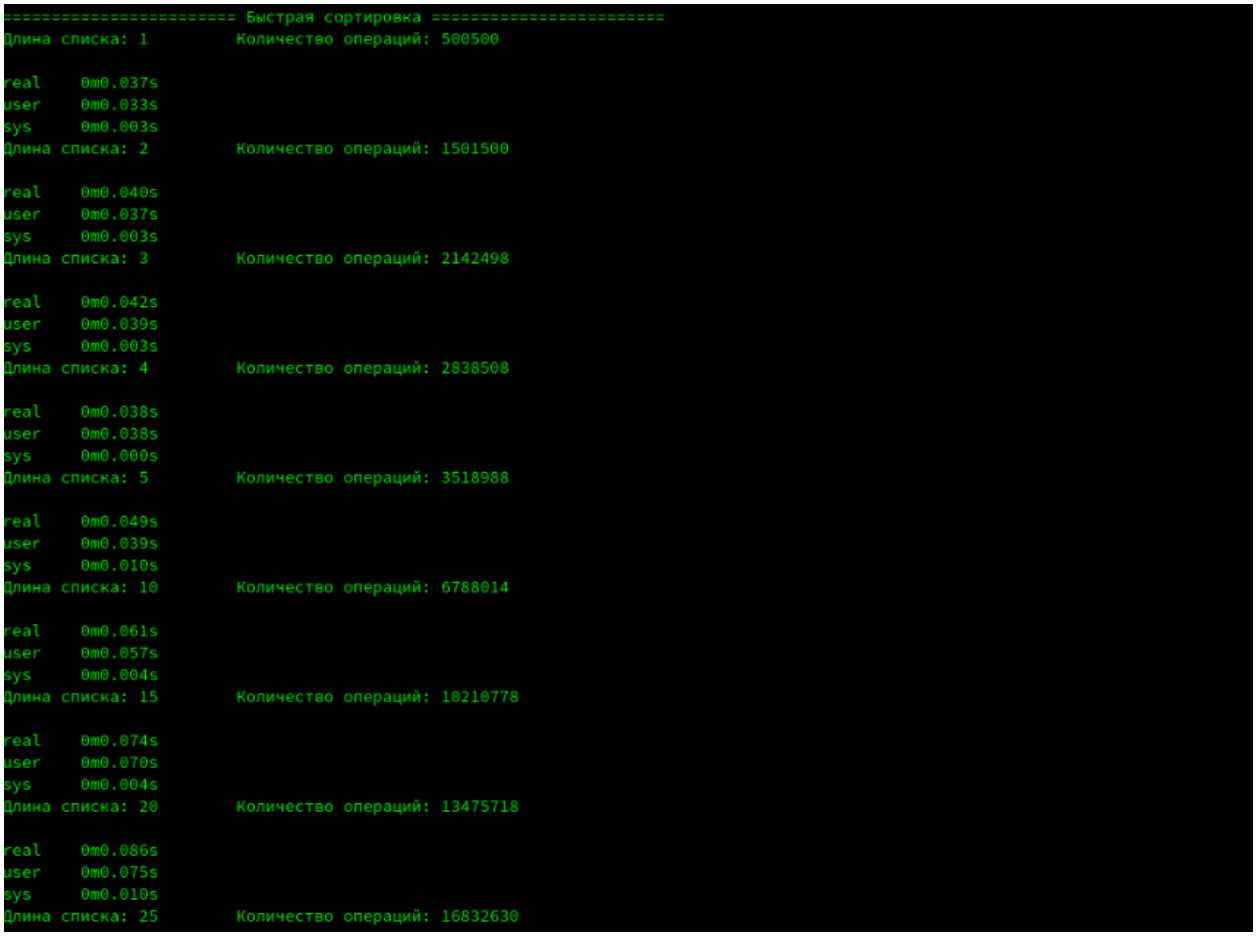

Рисунок 2.6 – Работа алгоритма Быстрой сортировки

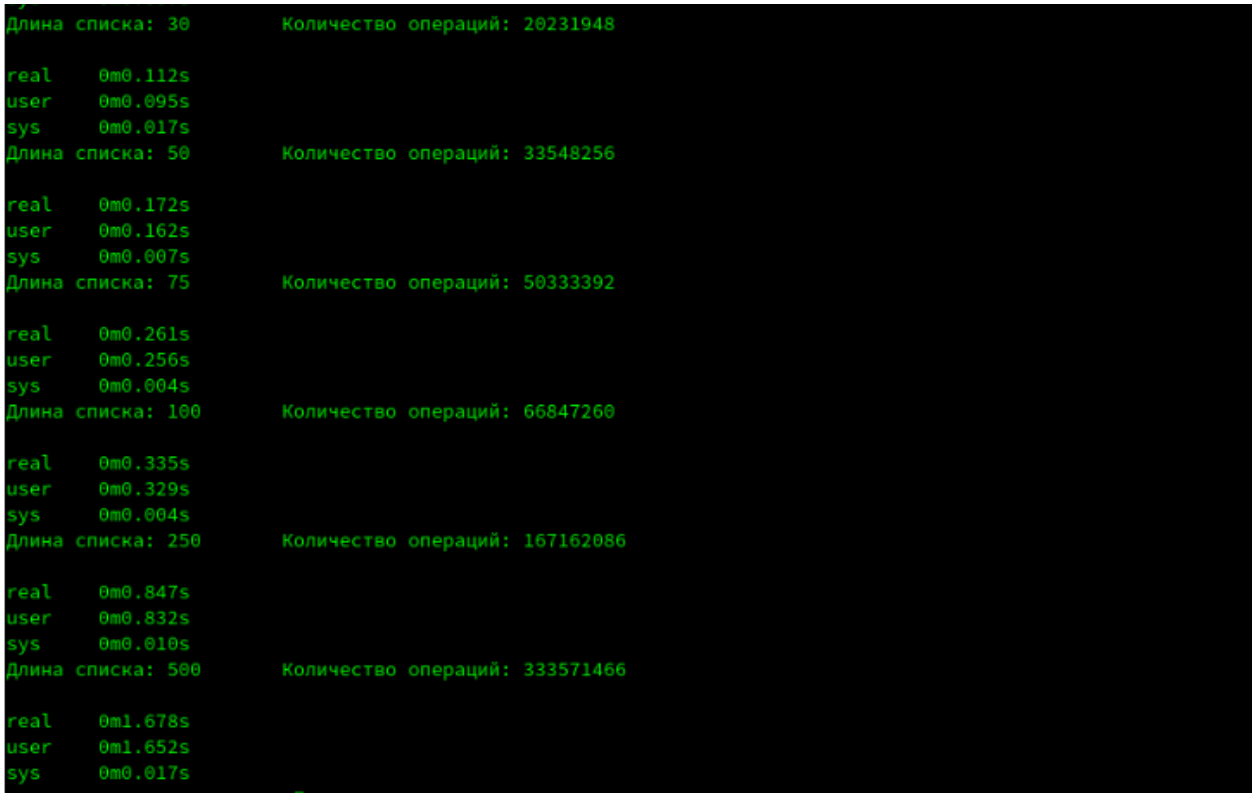

Рисунок 2.7 – Работа алгоритма Быстрой сортировки

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.

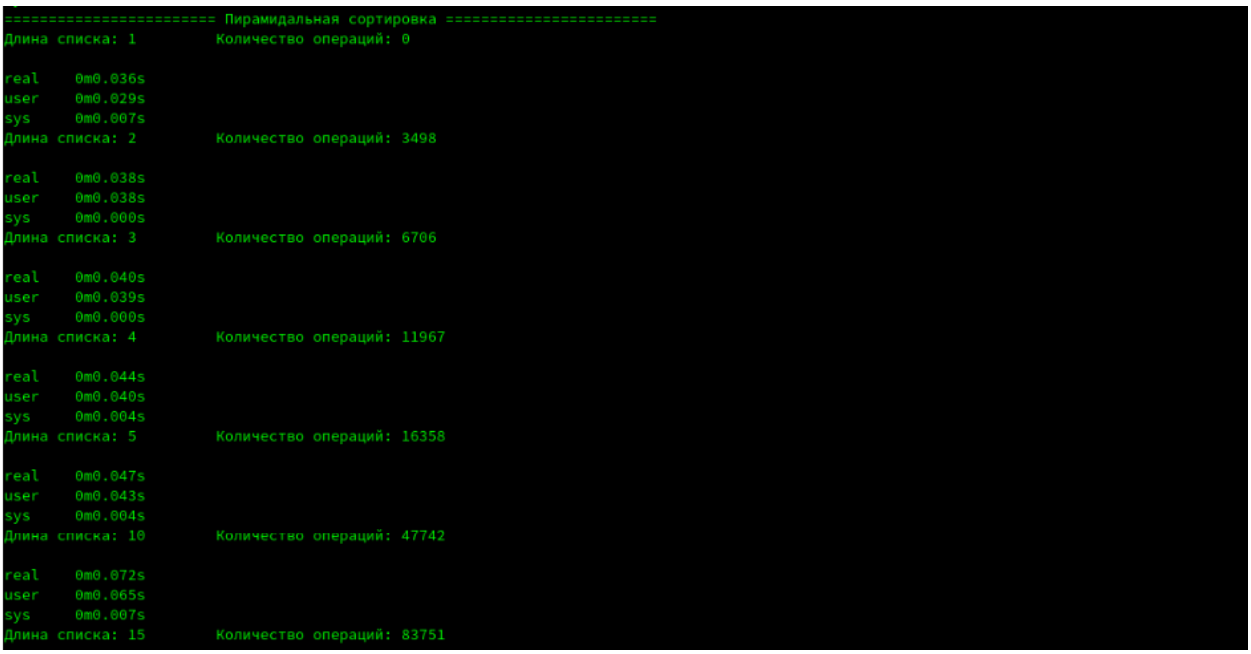

Рисунок 2.8 – Работа алгоритма Пирамидальной сортировки

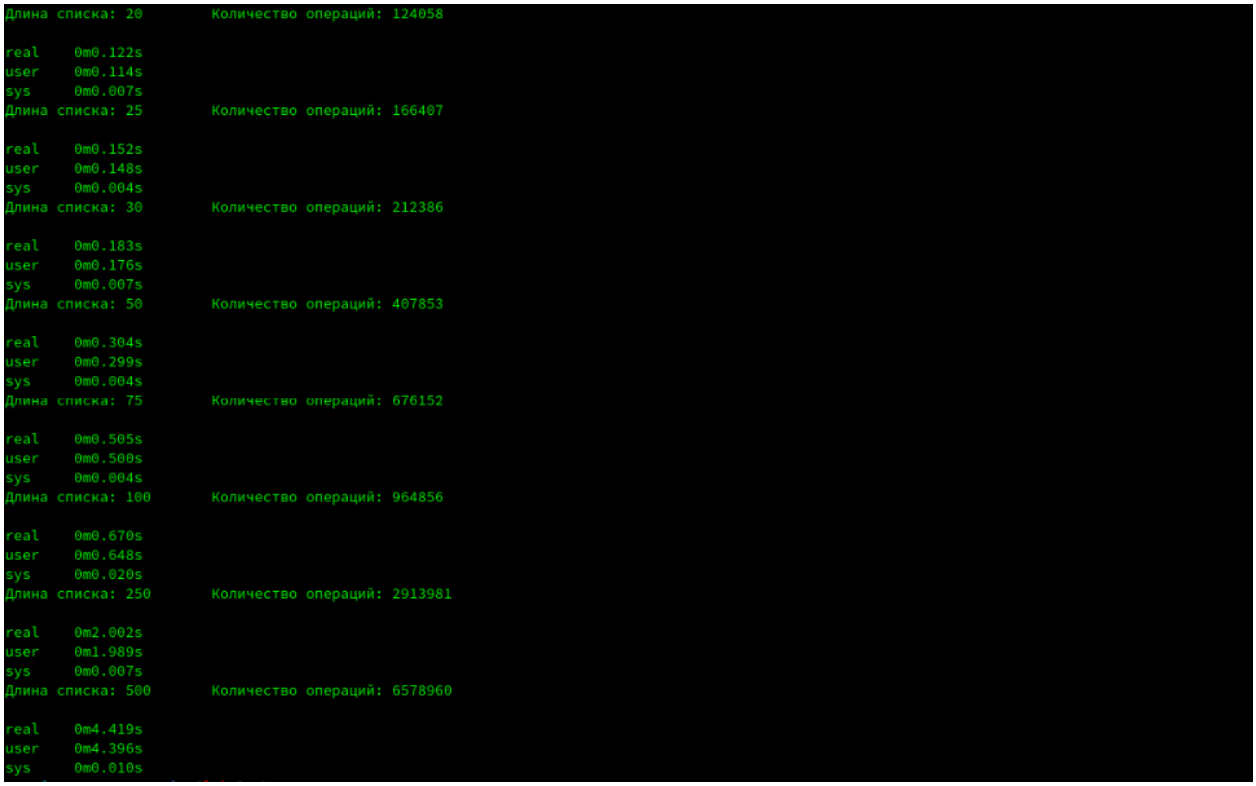

Рисунок 2.9 – Работа алгоритма Пирамидальной сортировки

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

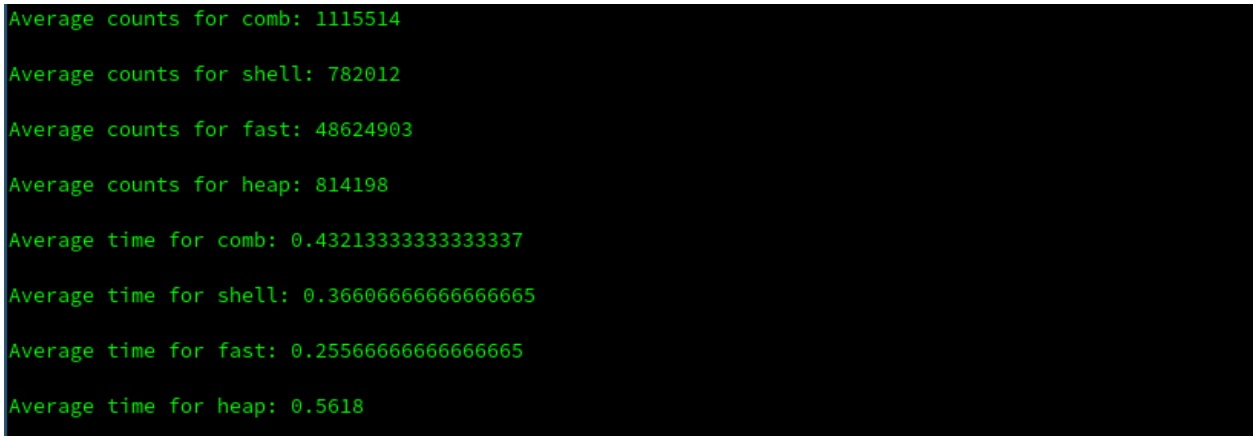

Рисунок 2.10 – Среднее время и среднее количество операций для всех алгоритмов

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).

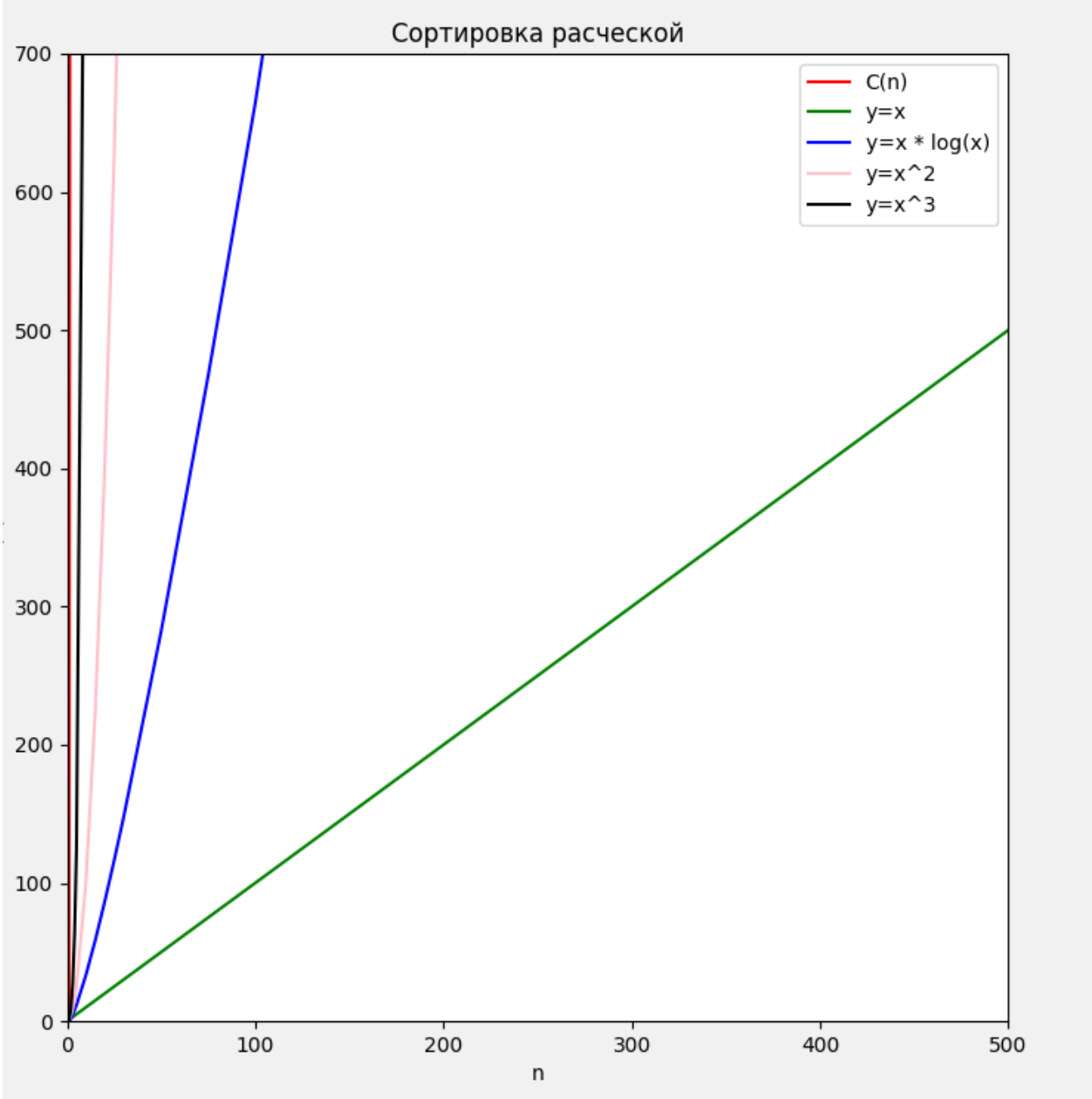

Рисунок 2.11 – График зависимости количества операций от длины массива для сортировки Расческой

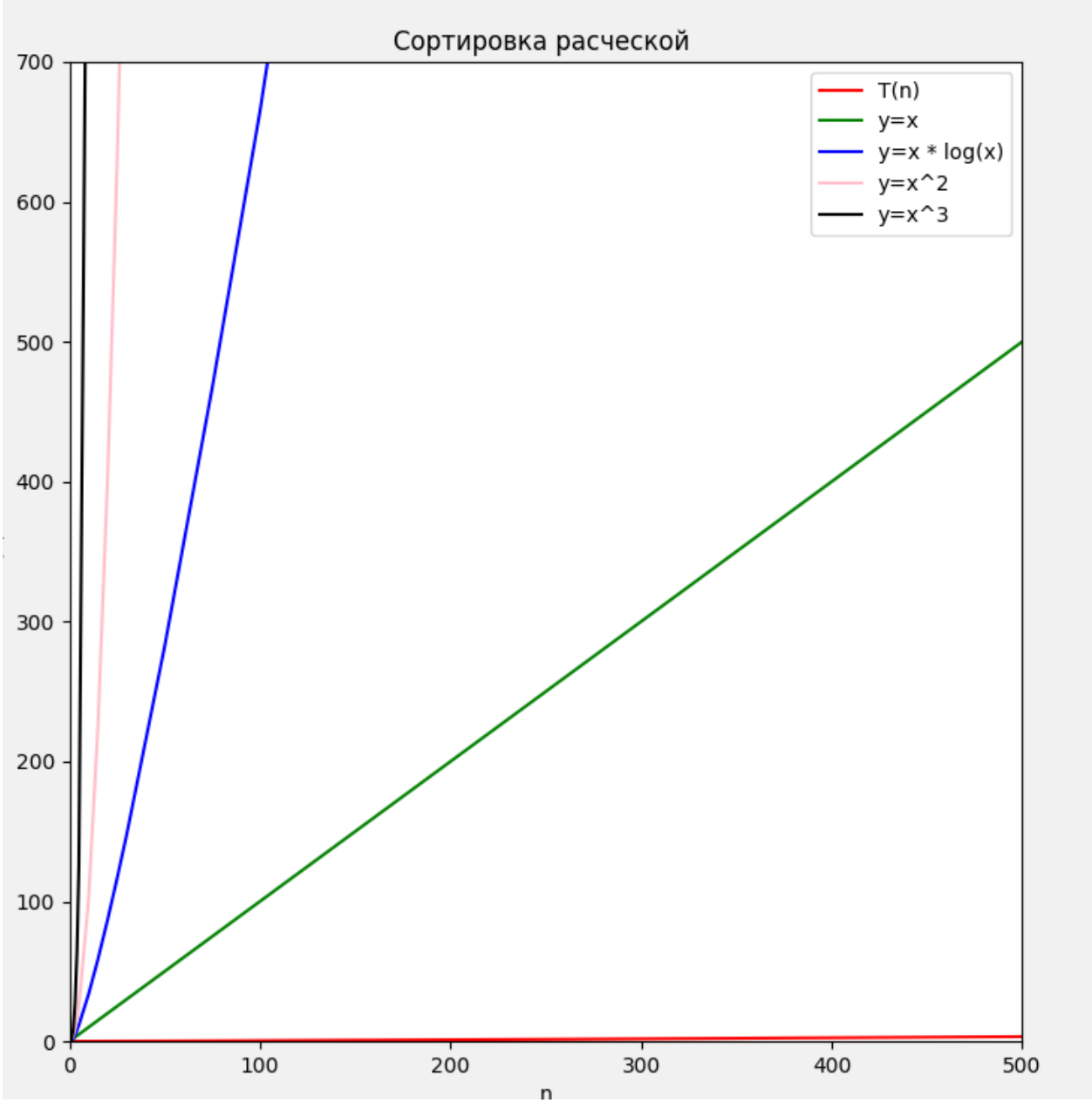

Рисунок 2.12 – График зависимости времени от длины массива для сортировки Расческой

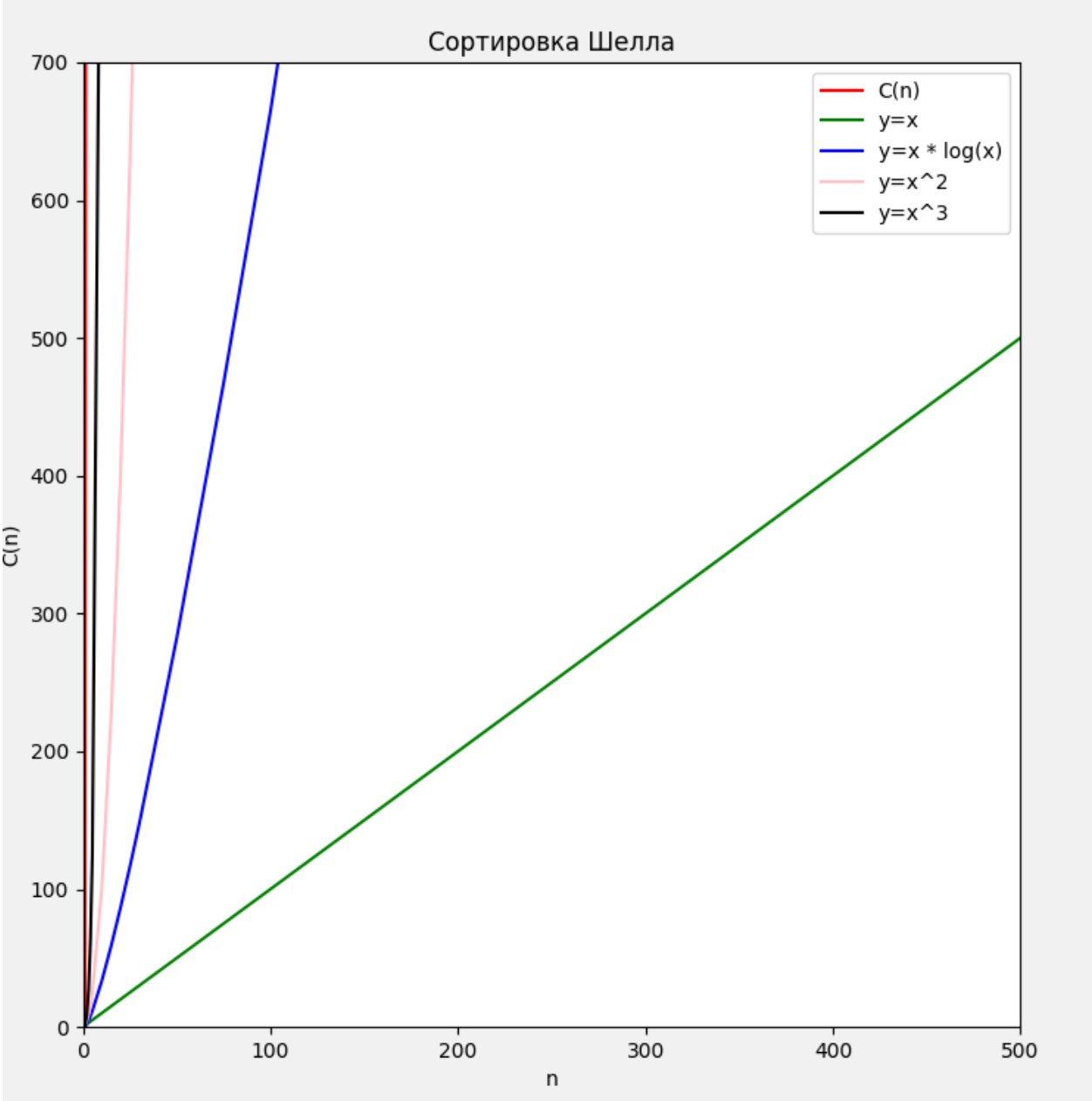

Рисунок 2.13 – График зависимости количества операций от длины массива для сортировки Шелла

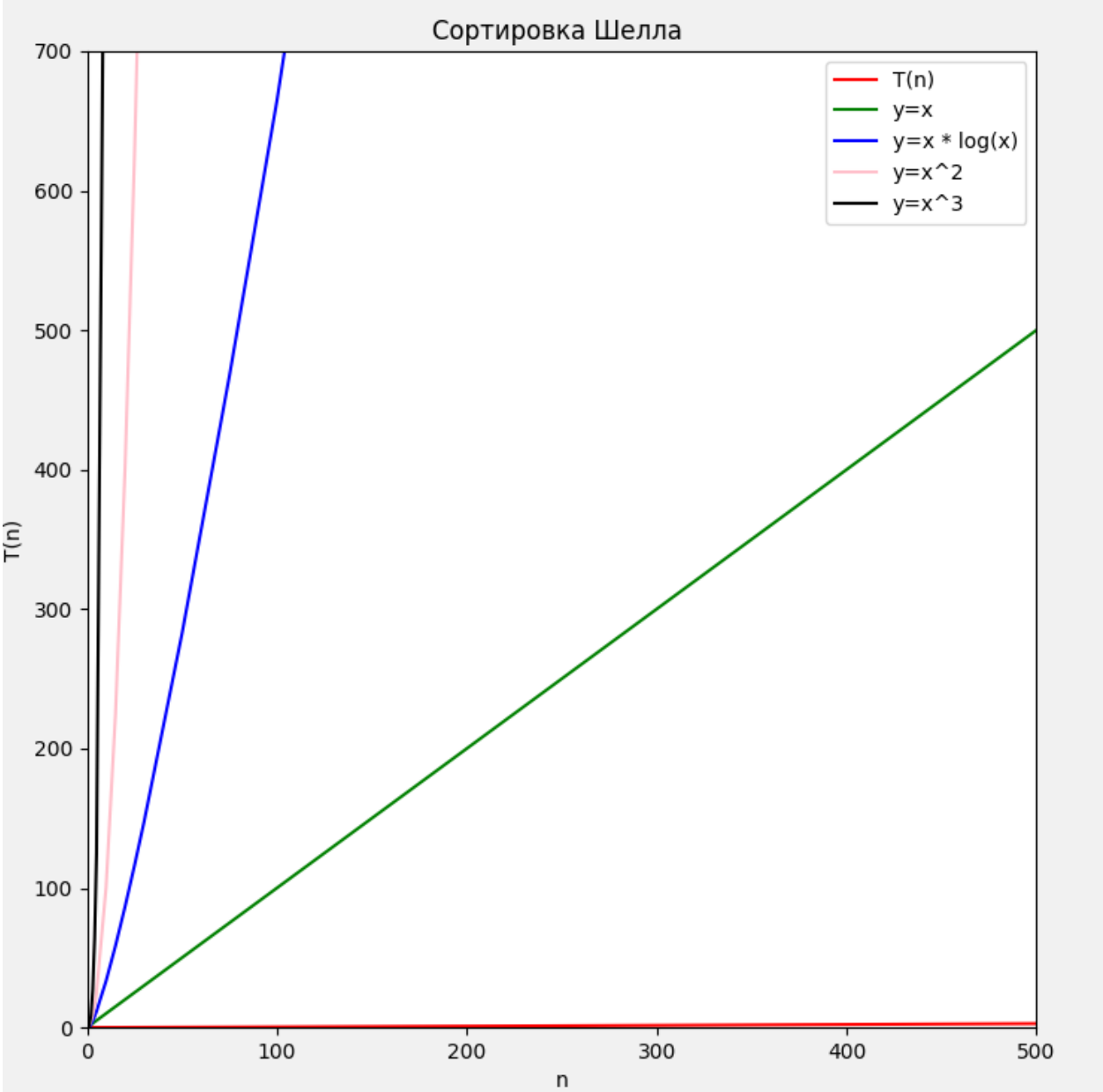

Рисунок 2.14 – График зависимости времени от длины массива для сортировки Шелла

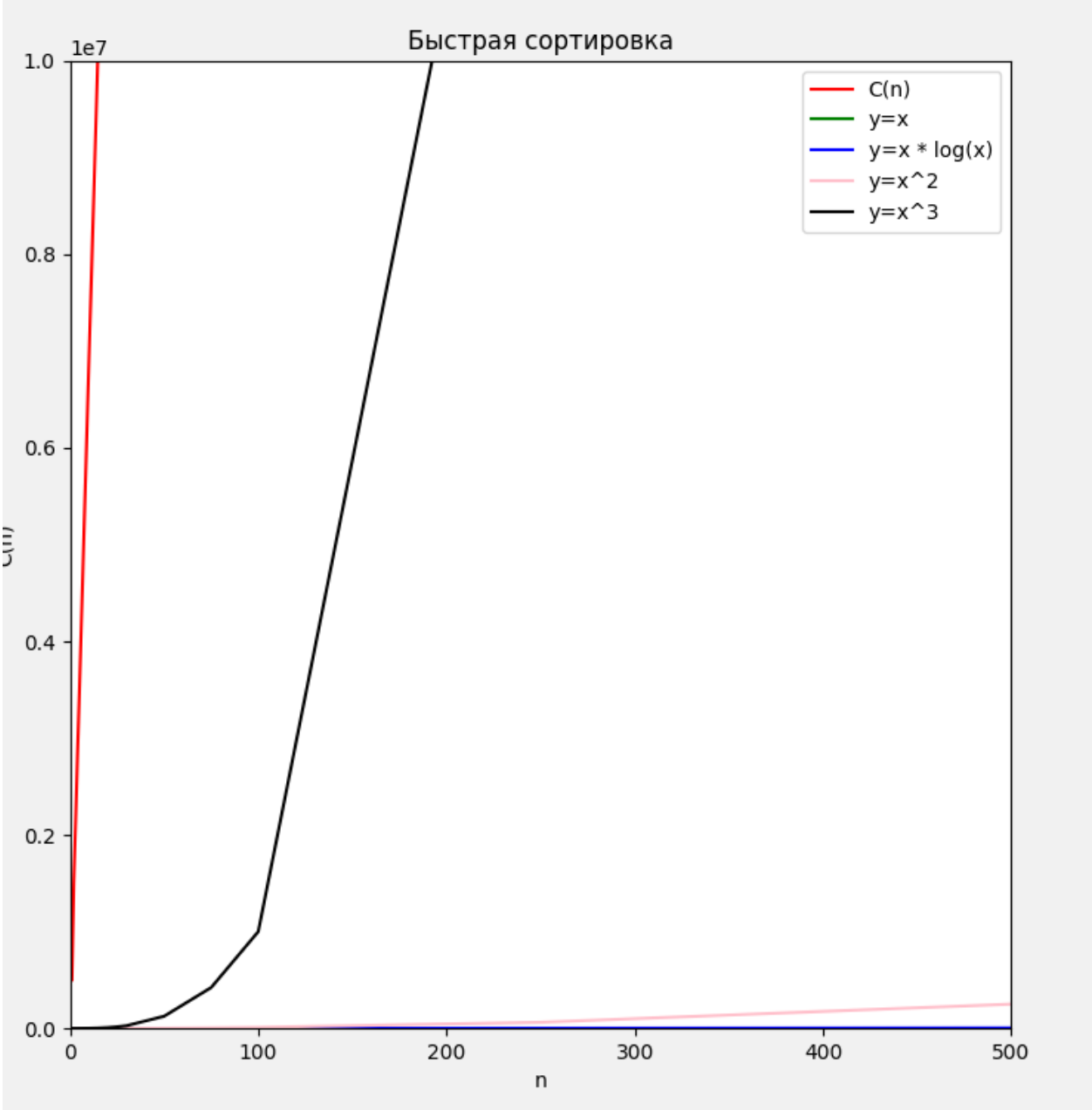

Рисунок 2.15 – График зависимости количества операций от длины массива для Быстрой сортировки

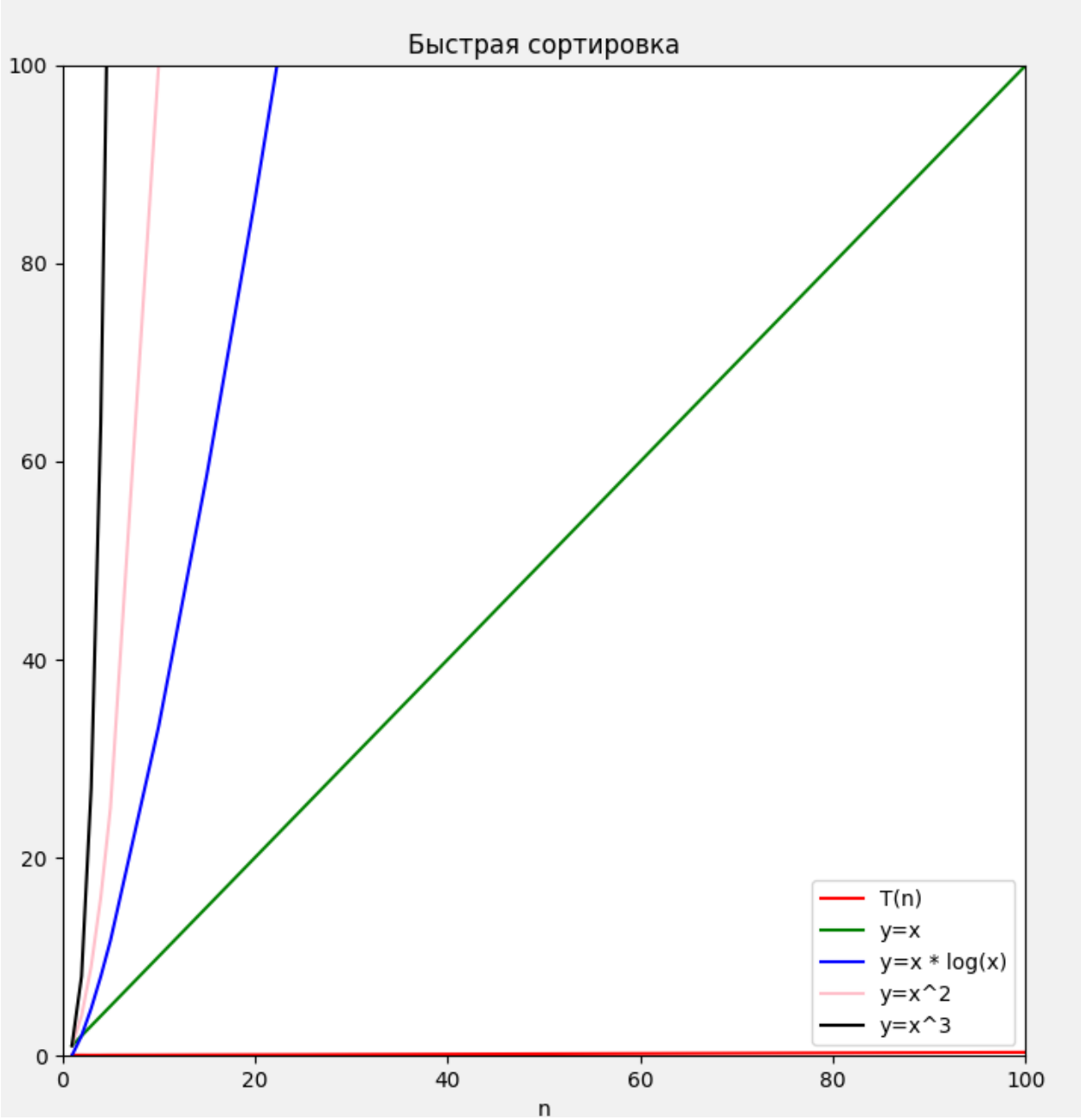

Рисунок 2.16 – График зависимости времени от длины массива для Быстрой сортировки

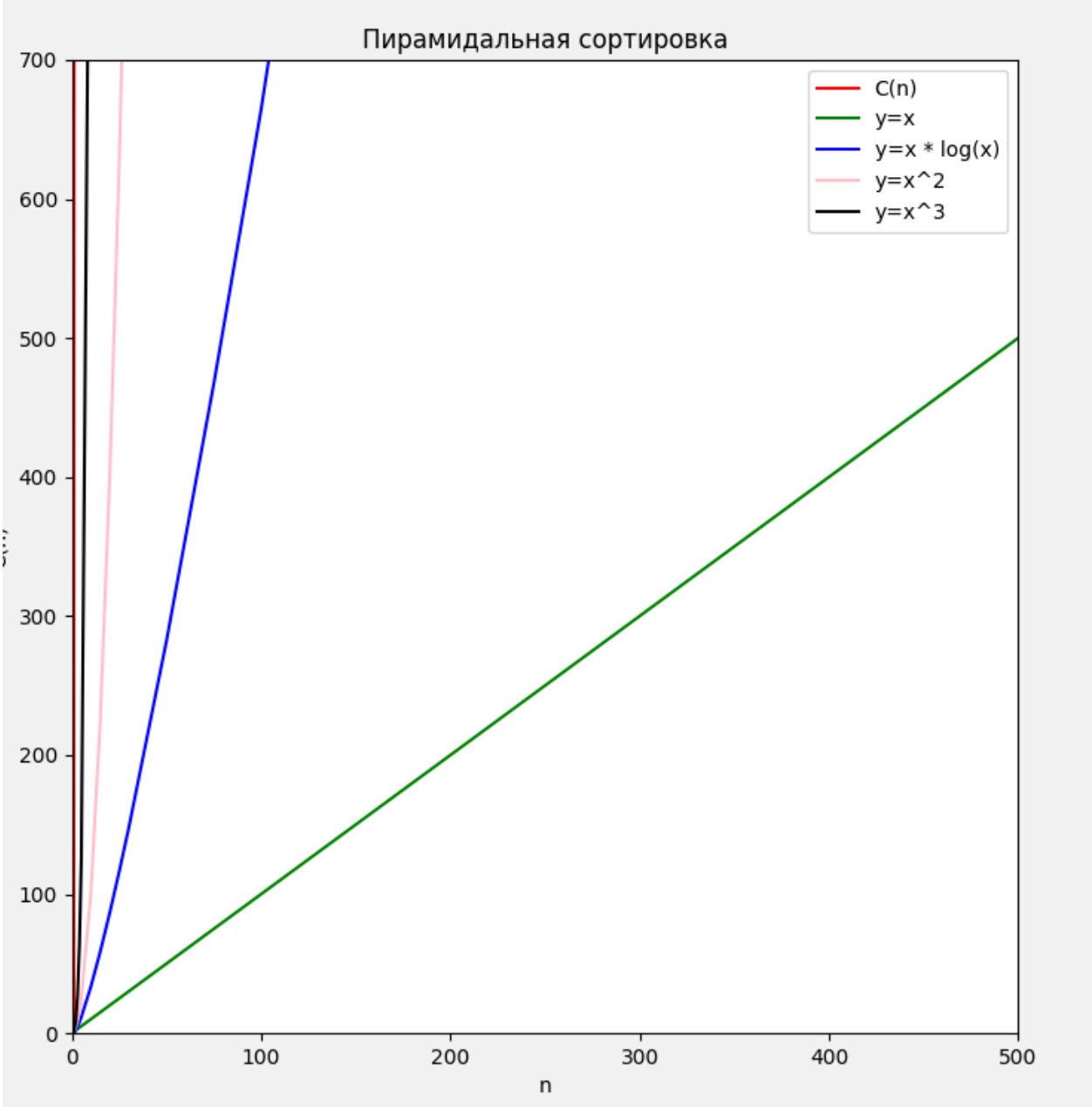

Рисунок 2.17 – График зависимости количества операций от длины массива для Пирамидальной сортировки

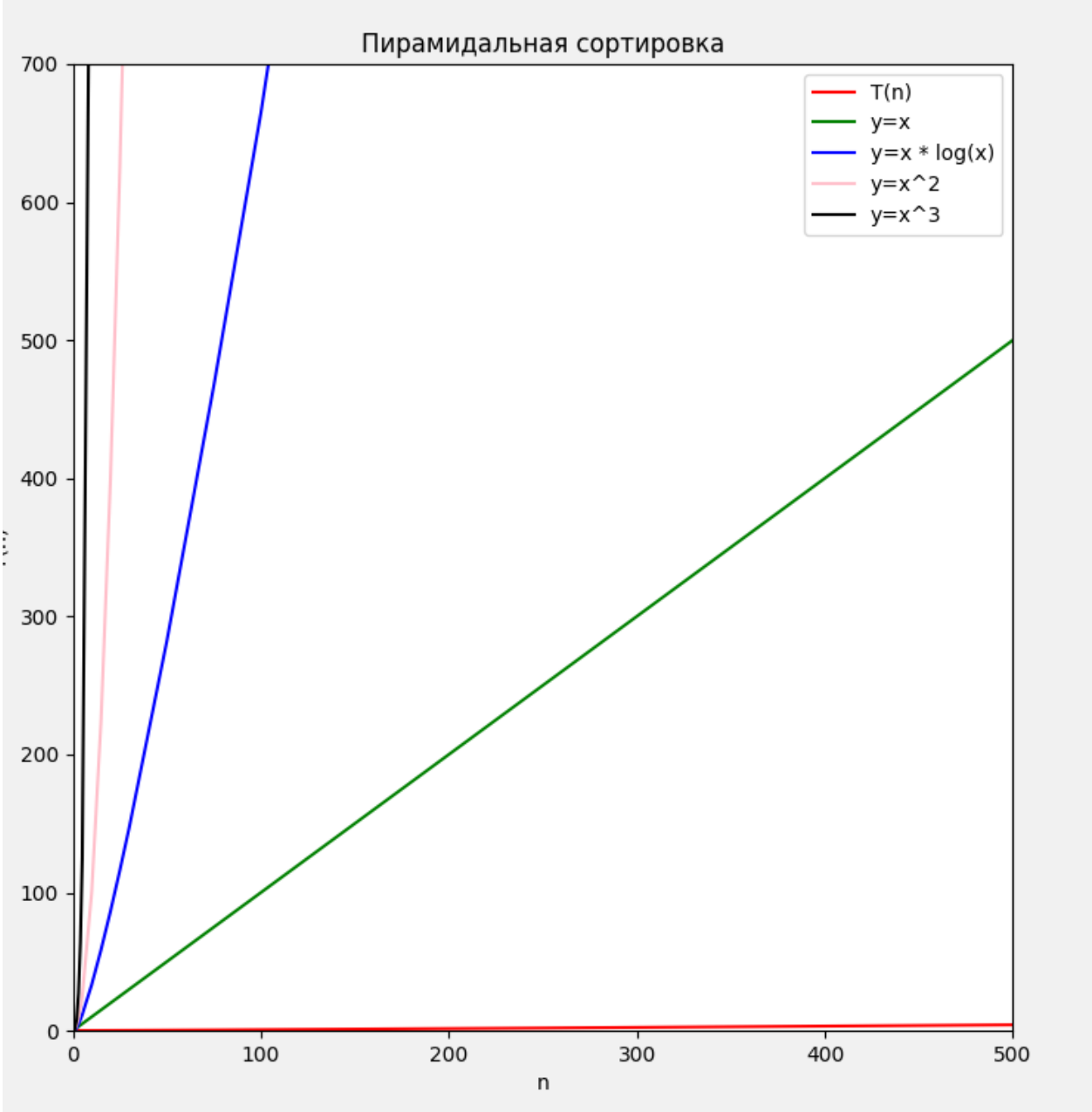

Рисунок 2.18 – График зависимости времени от длины массива для Пирамидальной сортировки

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

$$O(n * \log(n)) \tag{2.1}$$

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

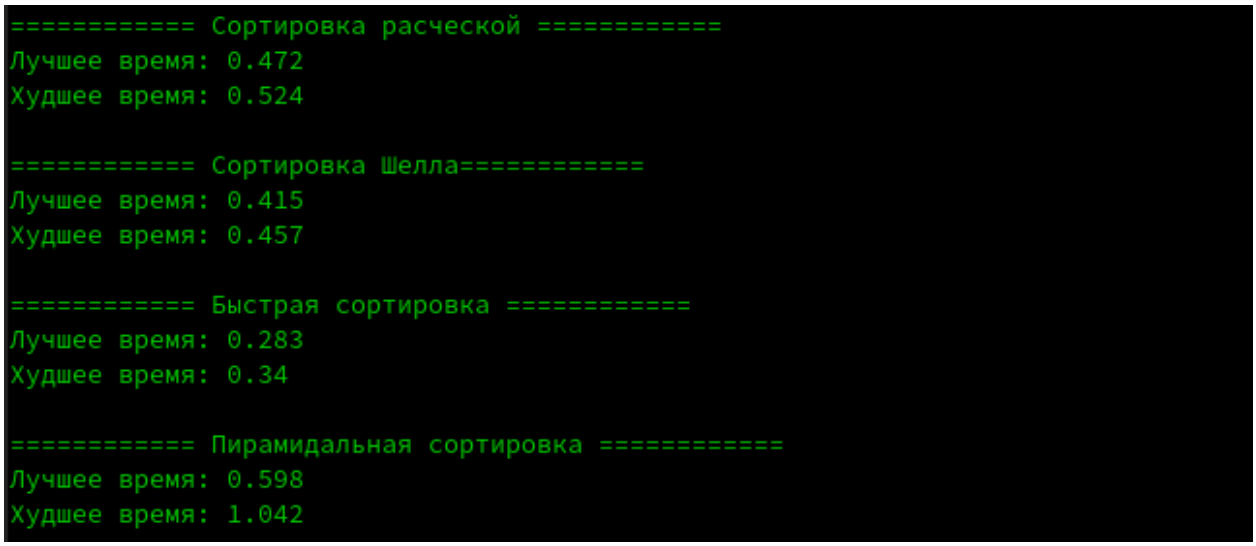

Рисунок 2.19 – Лучшее и худшее время работы каждого алгоритма

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 из тысячи списков
            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 элем.
        max_arr = [i for i in lst[1:] if i >= pivot]  # список с элементами больше опорного элем.
        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()
```