ОЦЕНКА АЛГОРИТМИЧЕСКОЙ СЛОЖНОСТИОтчет 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](#_Toc39120125)

[2 Ход работы 4](#_Toc39120126)

[3 Заключение 19](#_Toc39120127)

[Приложение А (Обязательное) Листинг программы для генерации списков 20](#_Toc39120128)

[Приложение Б (Обязательное) Листинг программы сортировка расческой 21](#_Toc39120129)

[Приложение В (Обязательное) Листинг программы сортировки Шелла 23](#_Toc39120130)

[Приложение Г (Обязательное) Листинг программы быстрой сортировки 26](#_Toc39120131)

[Приложение Д (Обязательное) Листинг программы пирамидальной сортировки 28](#_Toc39120132)

# 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)

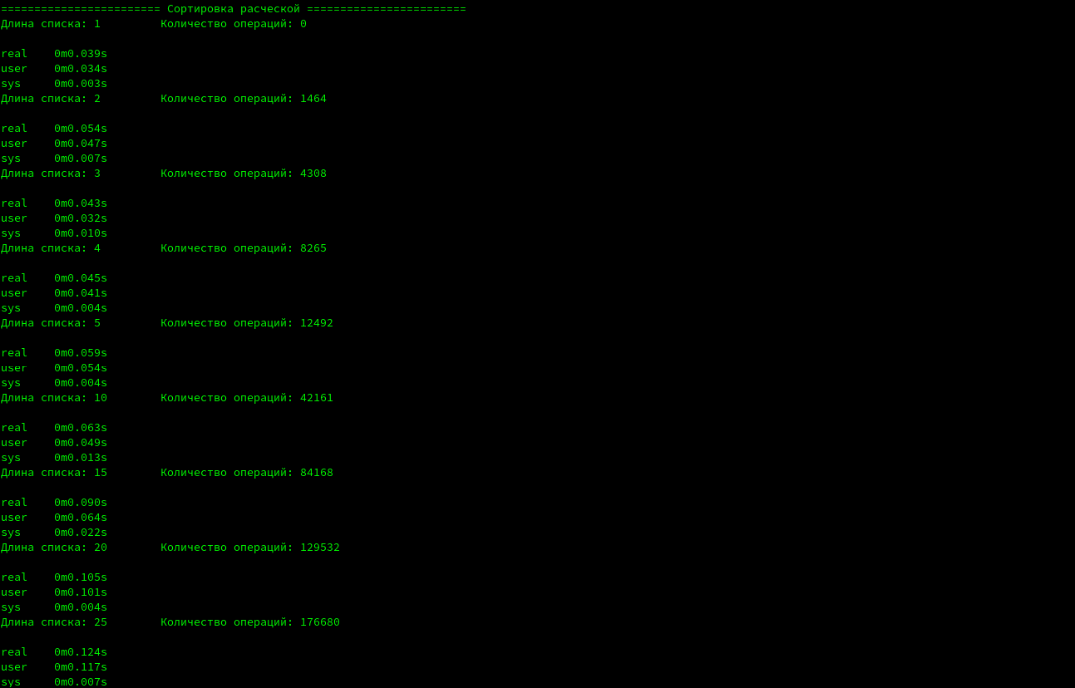


Рисунок 2.2 – Количество операций и затраченное время на сортировку Расческой

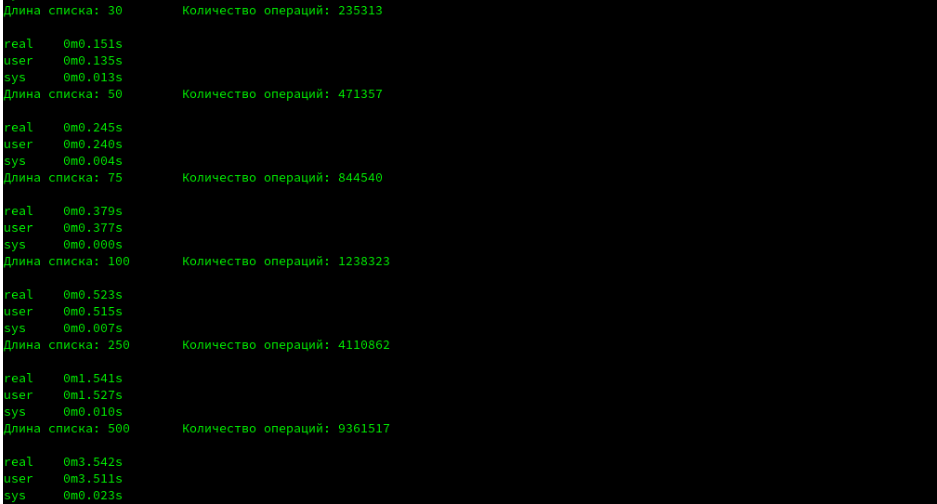


Рисунок 2.3 – Количество операций и затраченное время на сортировку Расческой

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

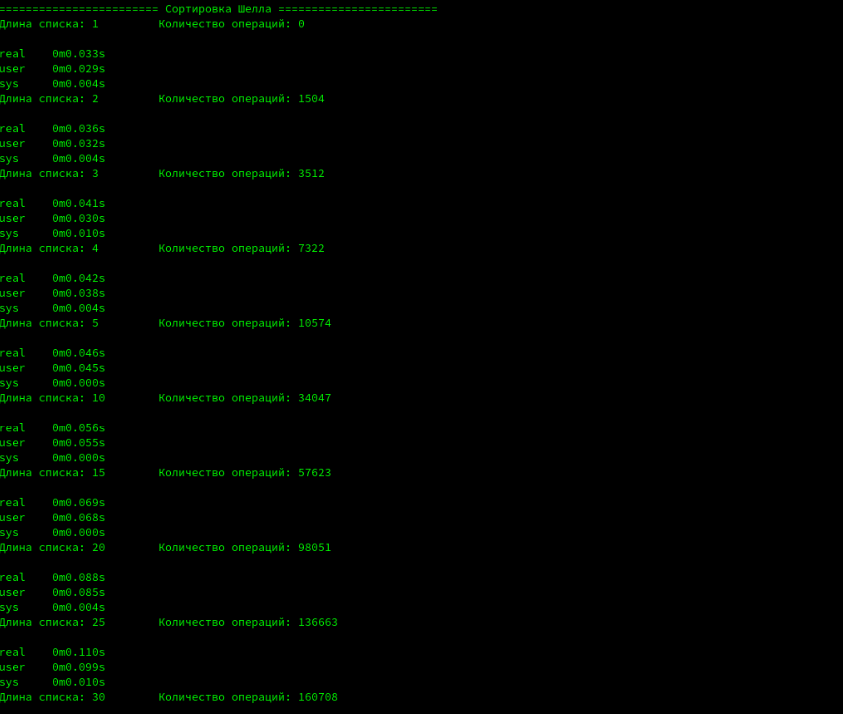


Рисунок 2.4 – Количество операций и затраченное время на сортировку Шеллом

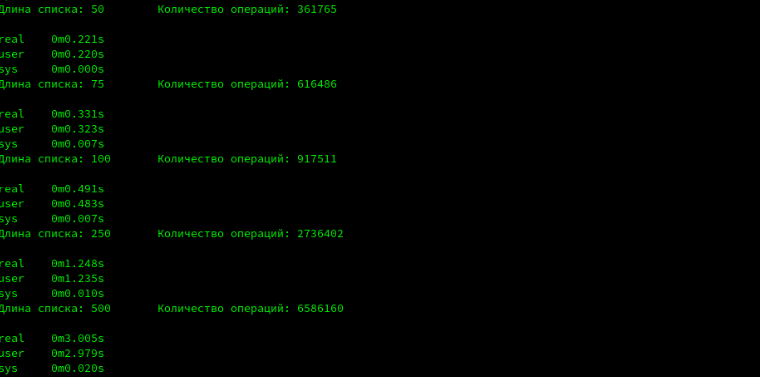


Рисунок 2.5 – Количество операций и затраченное время на сортировку Шеллом

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.

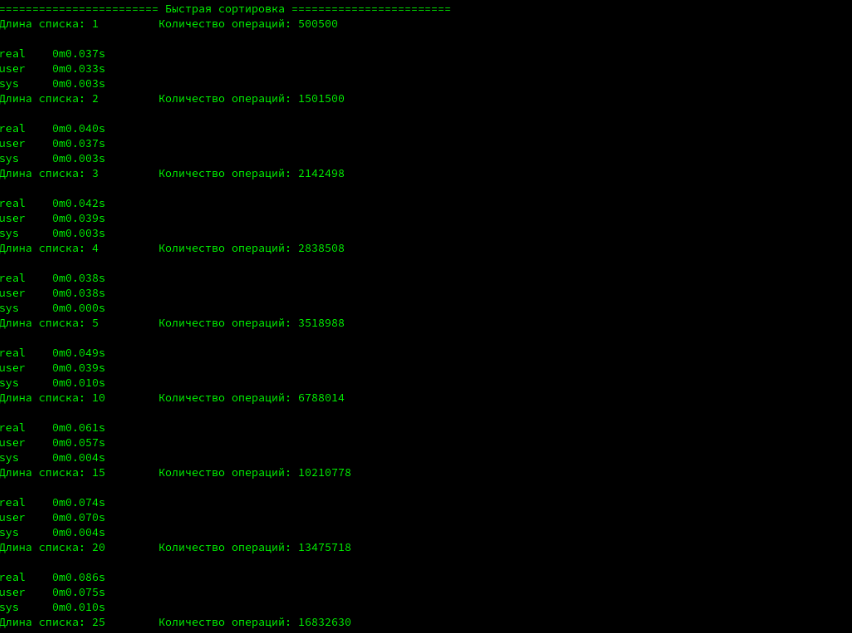


Рисунок 2.6 – Работа алгоритма Быстрой сортировки

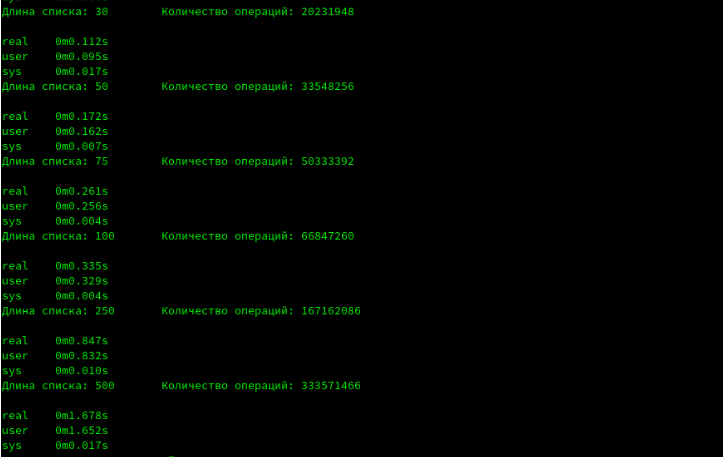


Рисунок 2.7 – Работа алгоритма Быстрой сортировки

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.

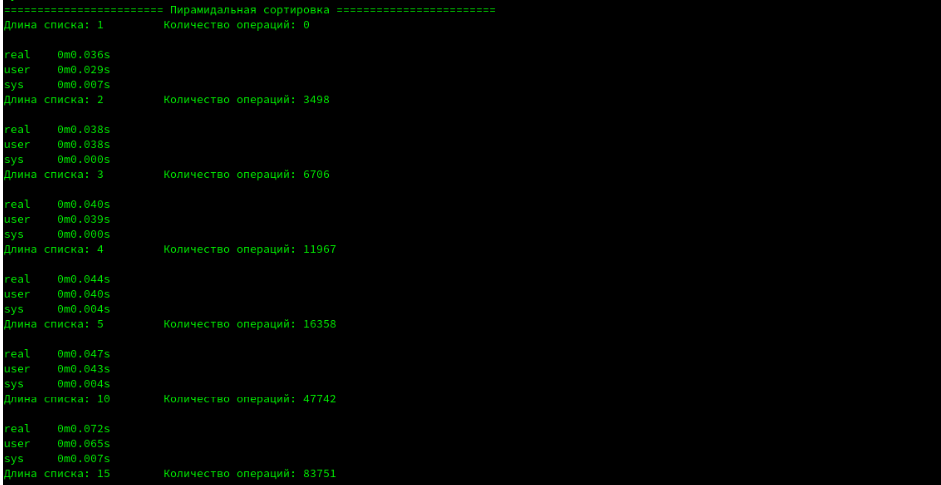


Рисунок 2.8 – Работа алгоритма Пирамидальной сортировки

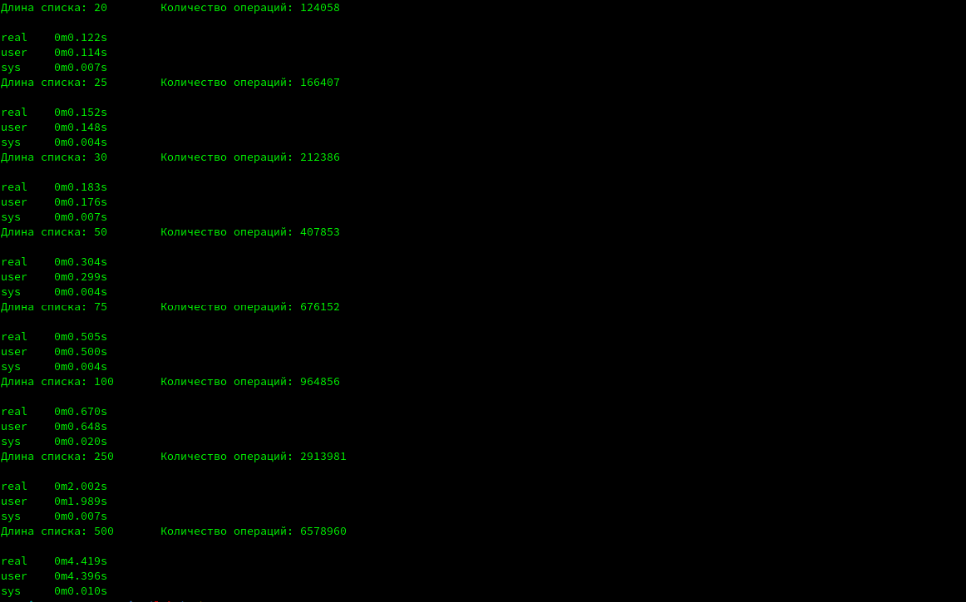


Рисунок 2.9 – Работа алгоритма Пирамидальной сортировки

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

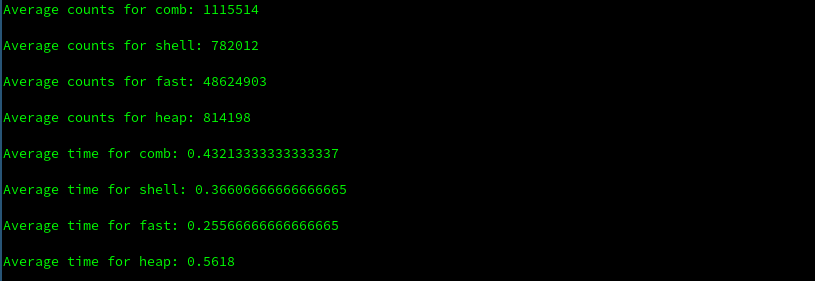


Рисунок 2.10 – Среднее время и среднее количество операций для всех алгоритмов

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).

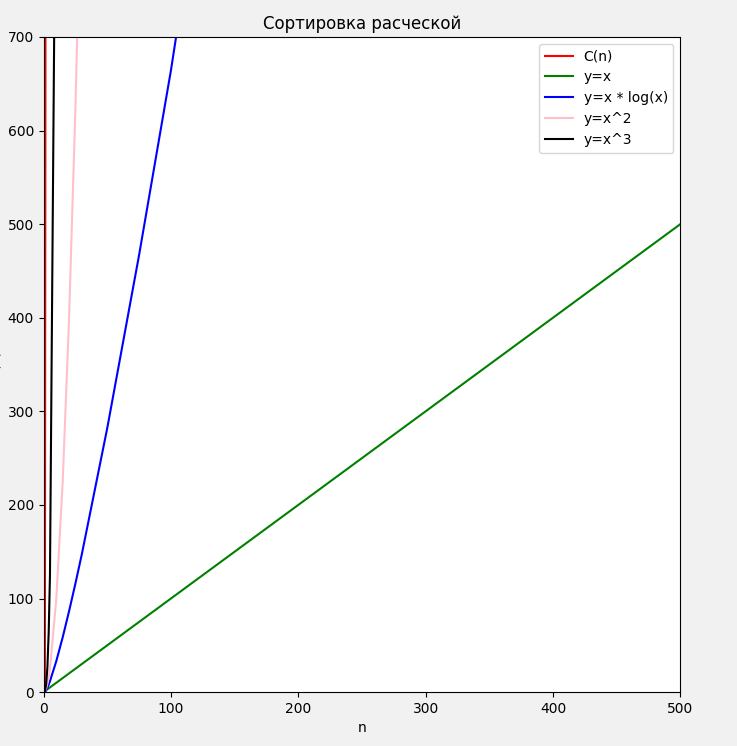


Рисунок 2.11 – График зависимости количества операций от длины массива для сортировки Расческой

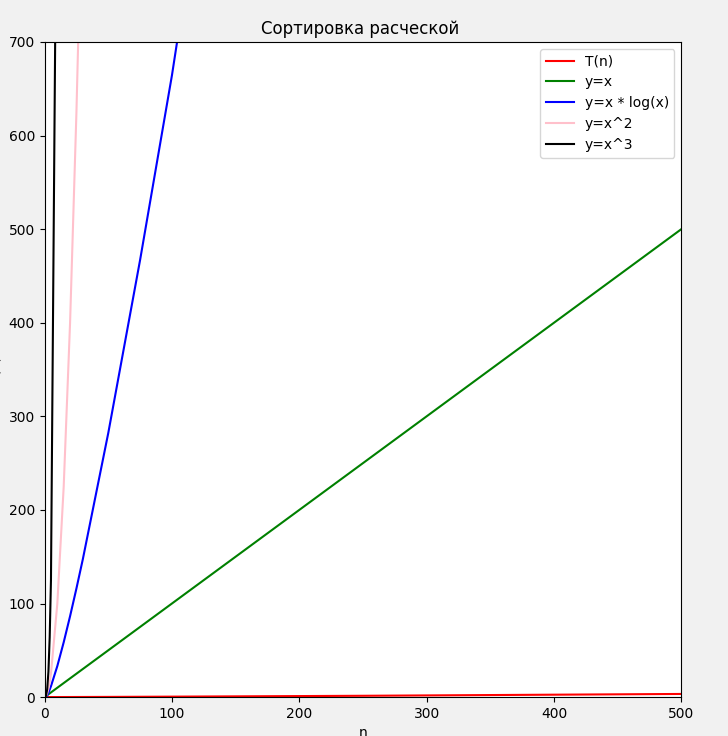


Рисунок 2.12 – График зависимости времени от длины массива для сортировки Расческой

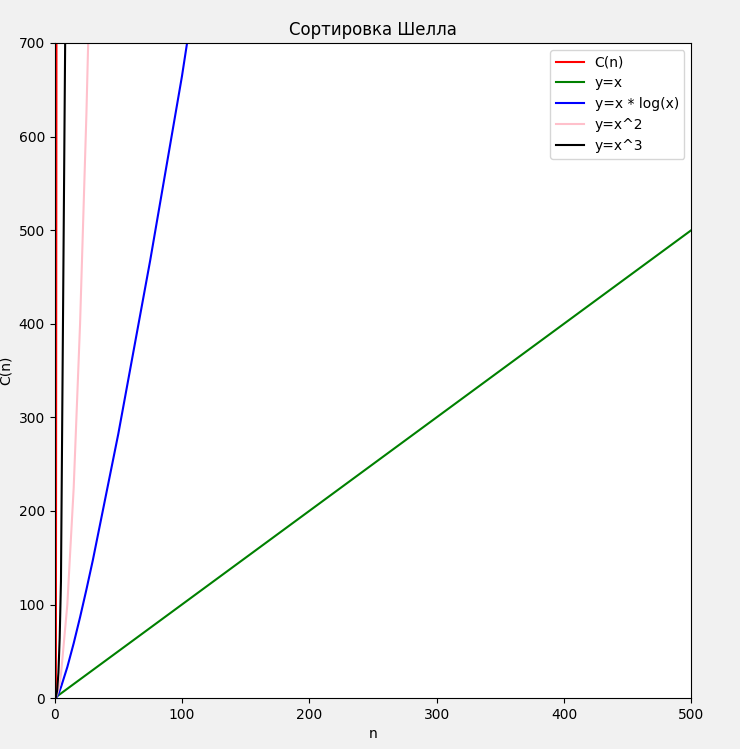


Рисунок 2.13 – График зависимости количества операций от длины массива для сортировки Шелла

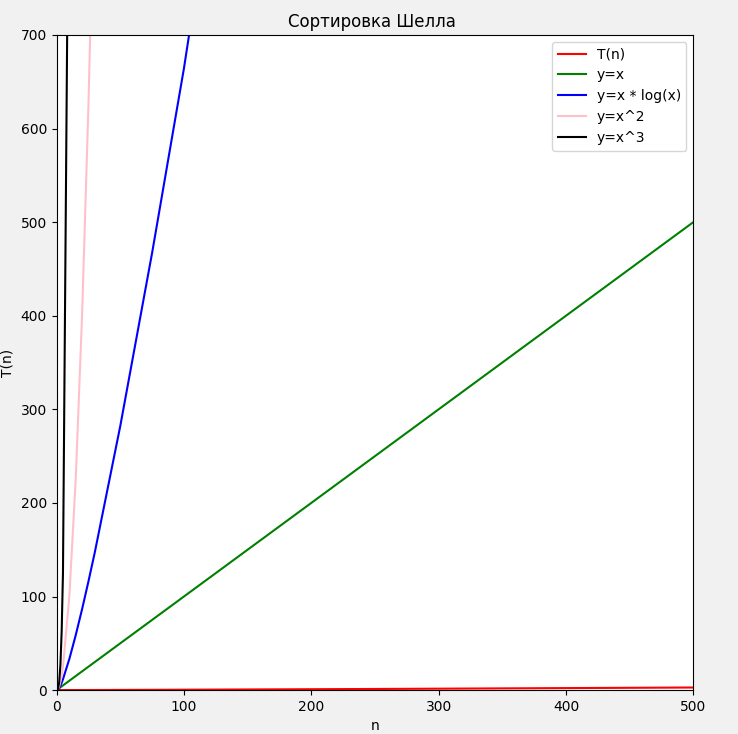


Рисунок 2.14 – График зависимости времени от длины массива для сортировки Шелла

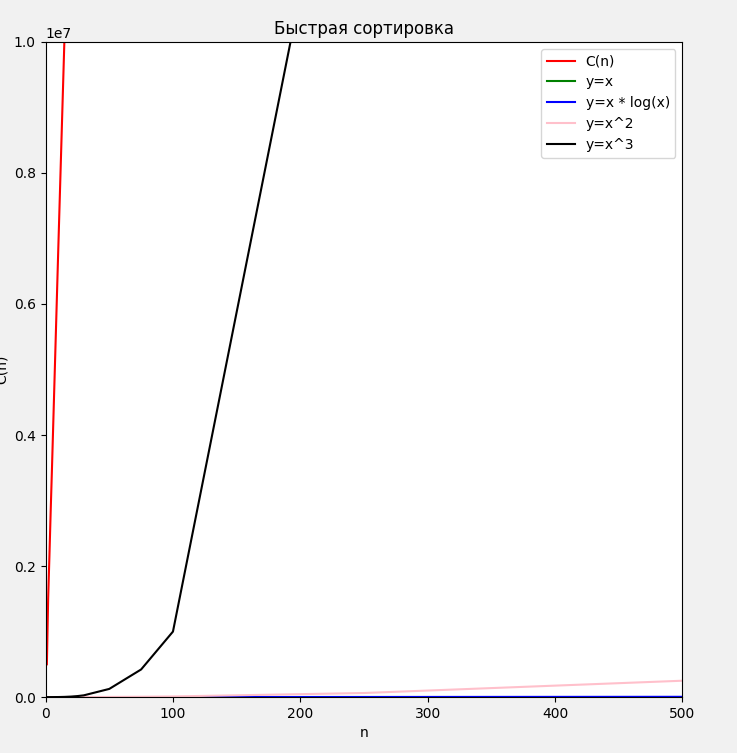


Рисунок 2.15 – График зависимости количества операций от длины массива для Быстрой сортировки

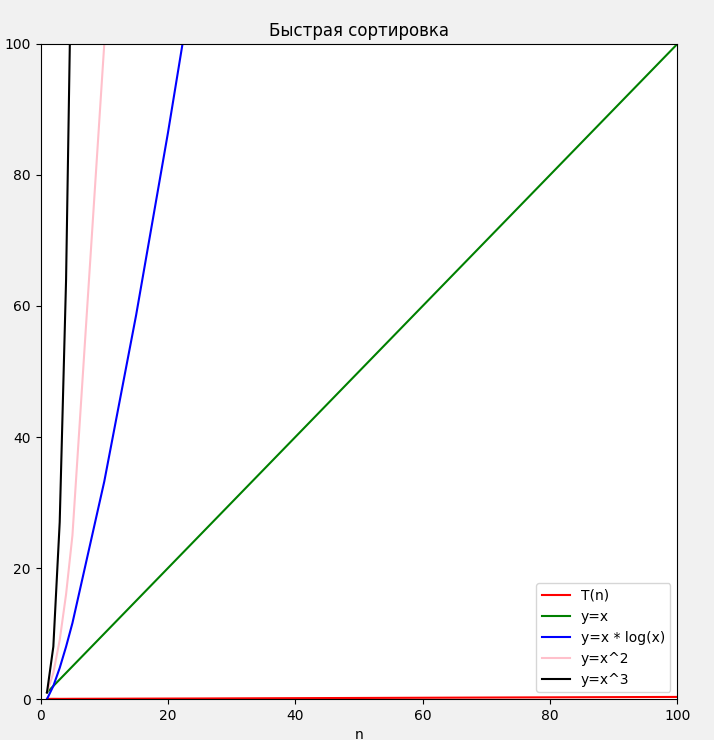


Рисунок 2.16 – График зависимости времени от длины массива для Быстрой сортировки

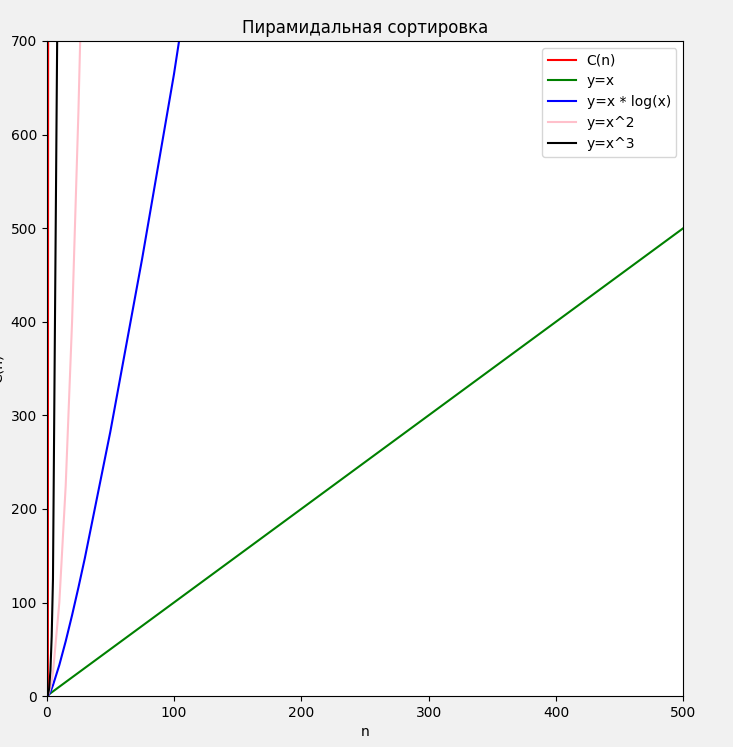


Рисунок 2.17 – График зависимости количества операций от длины массива для Пирамидальной сортировки

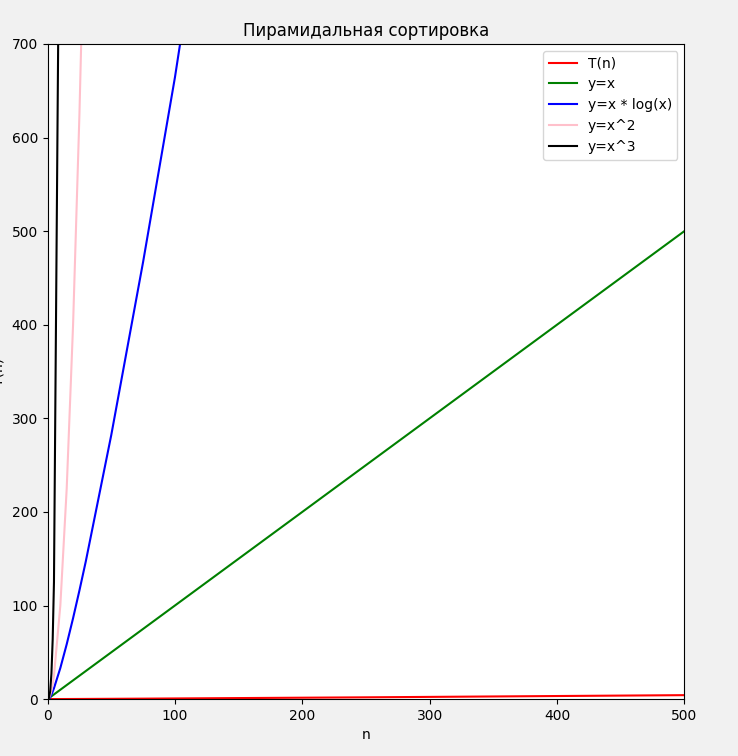


Рисунок 2.18 – График зависимости времени от длины массива для Пирамидальной сортировки

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

(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)

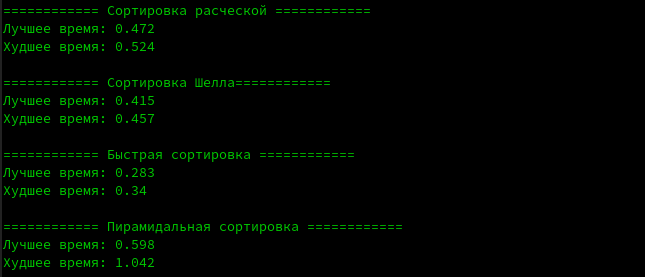


Рисунок 2.19 – Лучшее и худшее время работы каждого алгоритма

# 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()