МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 42

ОТЧЁТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

| ассистент |  |  |  | С.Ю. Гуков |
| --- | --- | --- | --- | --- |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

| ОТЧЁТ О ЛАБОРАТОРНОЙ РАБОТЕ №1 |
| --- |
| Сортировки. Определение сложности алгоритма. |
| по курсу: |
| АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ |
|  |

РАБОТУ ВЫПОЛНИЛ

| СТУДЕНТ гр. № | 4329 |  |  |  | Д.С. Шаповалова |
| --- | --- | --- | --- | --- | --- |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

ОГЛАВЛЕНИЕ

[Цель работы 3](#_30j0zll)

[Постановка задачи 3](#_1fob9te)

[Схема алгоритма решения 4](#_3znysh7)

[Полное описание реализованной функции 4](#_gqc1nhfmq022)

[Листинг программы 5](#_2et92p0)

[Результат выполнения программы. 5](#_tyjcwt)

[ВЫВОДЫ 6](#_3dy6vkm)

# Цель работы

Изучить работу алгоритмов сортировок. Научиться определять сложность алгоритмов.

# Постановка задачи

Задание: Написать программу, для сортировки и анализа текстовых данных. В соответствии с параметрами варианта задания нужно реализовать сортировку слов в тексте и провести его простейший анализ. Вариант задания приведён в таблице 1.

Таблица 1. Индивидуальное задание

| № | Текст задания | Вход | Выход |
| --- | --- | --- | --- |
| 17 | Быстрая сортировка.  Кириллица. По алфавиту. По возрастанию. Игнорировать числа.  Путь либо название считываемого файла вводится через консоль. Символы игнорируются.  Изобразить (график/гистограмма/диаграмма) зависимость времени выполнения сортировки от количества слов в тексте.  Оценить и обосновать сложность получившегося алгоритма в O-нотации | Файл input.txt с исходным текстом | Файл results.txt с отсортированными словами. И файл analysis.txt с анализом текста. |

# Схема алгоритма решения

1. Считать текст как массив слов, игнорируя числа.
   1. Заменяем числа и знаки препинания на ничего.
   2. “Сплитуем” текст на слова по пробелу
2. “Быстро” сортируем.
3. Записываем данные в файлик с анализом.
   1. Собираем статистику - сколько слов начинаются на каждую букву алфавита.

# Полное описание реализованной функции

## Текст => Массив слов:

* 1. Считываем текст из файла input.txt в папке с программой
  2. Заменяем числа и знаки препинания на ничего.
  3. “Сплитуем” текст на слова по пробелу.

## Сортировка:

### *1 функция - Разделение массива.*

Выбираем опорный элемент, с которым будем сравнивать другие элементы подмассива: меньше - влево, больше - вправо.

Переменная part инициализируется значением start. На этом индексе будет располагаться опорный элемент после разделения массива.

Проходим циклом for каждый элемент массива, начиная с элемента после опорного.

Проверяем, меньше или равно значение текущего элемента unsorted[i] значению опорного элемента unsorted[start]. Если условие истинно, увеличивается индекс part, указывая место для размещения элементов, которые меньше или равны опорному.

Меняем значения: текущий элемент unsorted[i] с элементом на индексе part. В итоге все элементы, которые меньше или равны опорному элементу будут на левой стороне массива.

В конце, опорный элемент (unsorted[start]) меняется местами с элементом на индексе part, тем самым перемещая опорный элемент на его конечное место в отсортированном массиве. => Все элементы массива, меньшие опорного стали слева, а большие справа.

Возвращаем индекс part, чтобы использовать его в следующих вызовах рекурсивной сортировки, тем самым определяя границы двух подмассивов для дальнейшей сортировки.

### *2 функция - запуск сортировки - quick\_sort*

Если end не задан, его значение устанавливается как индекс последнего элемента массива, что охватывает весь массив.

### *3 функция - подфункция - непосредственно сортировка - quick :*

Проверка базового случая: если start больше или равен end, сортировка завершена для этой части, и функция возвращает None.

Вызываем функцию partition для разделения массива с возвращением индекса опорного элемента.

Затем происходит рекурсивные вызовы для сортировки левой части массива (элементы слева от опорного элемента) и для правой.

Вызов и возврат результата выполнения функции quick, которая выполняет всю сортировку.

## Анализ:

Записываем в выходной файл analysis.txt в папке с программой: введённый текст, задание варианта, количество слов в тексте, время сортировки, статистику (количество слов на каждую букву алфавита).

### *4 функция - подсчёт количества слов на определённую букву алфавита:*

Проходимся по словам в массиве слов, смотрим какая буква 1, если она искомая, то записываем в переменную “количество” +1, в конце цикла выводим.

## Сложность сортировки

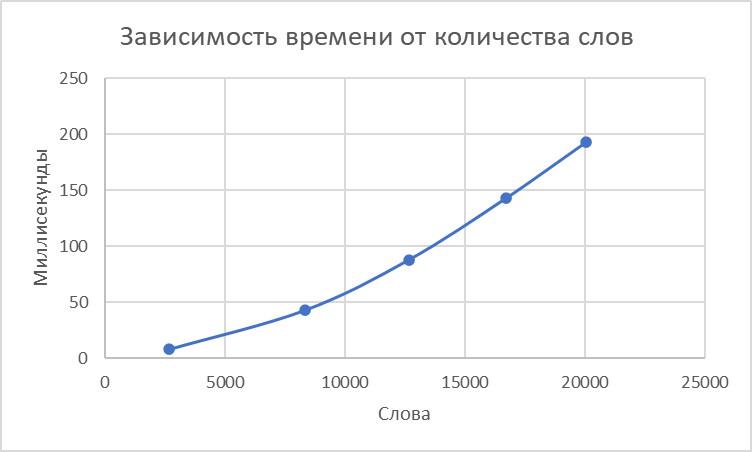


Рисунок 2.3. - График зависимости времени сортировки от объёма текста

По рисунку 2.3. мы можем “наглядно” наблюдать сложность алгоритма быстрой сортировки -

*Как к этому прийти?*

Сложность быстрой сортировки можно определить в зависимости от сценариев, комбинируя лучшие, средние и худшие случаи:

### *1. Худший случай*

Сценарий: Происходит, когда опорный элемент каждый раз оказывается на самом краю массива, например, массив отсортирован в противоположном порядке.

Сложность: O(n^2)

Каждое разбиение производится так, что один подмассив будет содержать n−1 элементов, а другой — 0. При этом каждый элемент сравнивается с опорным.

### *2. Средний случай*

Сценарий: Опорный элемент выбирается случайно, что приводит к более равному разбиению массива на подмассивы.

Сложность: O(nlog⁡n)

Если массив из N элементов делится на примерно две равные части, то минимальная глубина рекурсии (или количество разбиений, чтобы отсортировать массив) будет логарифмической по основанию 2.

То есть log2​N показывает, сколько раз мы можем разделить N пополам, прежде чем достигнем подмассива размером в 1.

Например, если у нас есть 16 элементов:

Делим на 2: 8.

Делим на 2: 4.

Делим на 2: 2.

Делим на 2: 1.

Потребовалось 4 деления, что соответствует log⁡216=4

### *3. Лучший случай*

Сценарий: Опорный элемент каждую итерацию делит массив ровно пополам.

Сложность: O(nlog⁡n)

Подобно среднему случаю, массив делится на две равные части, что приводит к логарифмическому количеству делений.

### *Общая оценка сложности быстрой сортировки:*

На каждом уровне рекурсии нужно сделать O(N) операций для перестановки и выбора опорного элемента.

Так как есть log⁡N уровней рекурсии и на каждом уровне требуется O(N) времени, общая сложность получается O(Nlog⁡N).

# Листинг программы

import time

import sys

sys.setrecursionlimit(2000) # Увеличиваем допустимую глубину рекурсии

# Запоминаем время начала работы

start\_time = time.time()

# Разделение массива

def partition(unsorted, start, end):

part = start # Опорный элемент - 0вой

for i in range(start + 1, end + 1): # end+1, чтобы включить последний эл-т (по-умолч. не включительно)

if unsorted[i] <= unsorted[start]: # если правый эл-т меньше опорного(он 0вой)

part += 1 # мы нашли ещё один элемент, который должен быть перемещён влево от опорного.

unsorted[i], unsorted[part] = unsorted[part], unsorted[i]

# все элементы, меньшие либо равные опорному, будут перемещены влево от него

unsorted[part], unsorted[start] = unsorted[start], unsorted[part] # опорный элемент, теперь на новом индексе

return part

# возвращает индекс опорного элемента, чтобы использовать его для следующего разбиения в рекурсивной сортировке

# Сортировка

def quick\_sort(unsorted, start=0, end=None):

if end is None: # только начали = охватываем весь массив

end = len(unsorted) - 1

def quick(unsorted, start, end):

if start >= end: # пришли в конец

return

part = partition(unsorted, start, end) # делим массив на меньшую часть и большую,

# относительно опорного эл-та, возвр. его индекс

# опорный элемент теперь где-то в серединке unsorted

quick(unsorted, start, part-1) # сортировка левой части массива — слева от опорного эл-та

quick(unsorted, part+1, end) # сортировка правой

return quick(unsorted, start, end) # возвращаем результат ф-и quick, которая выполняет всю сортировку

# считаем сколько слов на букву

def words\_per\_letter(letter, words):

count=0

for word in words:

if word[0] == letter:

count +=1

return count

# Считывание файла

# path\_of\_file = input()

path\_of\_file = 'input.txt' # для тестов

with open(path\_of\_file, 'r', encoding='utf-8') as in\_file:

text = in\_file.read()

input\_text = text

# Убираем числа и спец. символы

for cifra in '1234567890.,-—«"()„!?;':

text = text.replace(cifra, '')

# Разделяем текст на массив слов

words = text.lower().split()

# text.lower() - делаем заглавные и строчные буквы строчными

# Сортируем

quick\_sort(words, start=0, end=None)

# Считаем время сортировки

end\_time = time.time()

sort\_time\_sec = end\_time - start\_time

sort\_time\_mlsec = sort\_time\_sec\*1000

# Анализ

## Введённый текст

with open("analysis.txt", "w", encoding='utf-8') as anal\_file:

# Записываем строки в файл

anal\_file.write("Введённый текст: \*5 глав Преступления и Наказания\*\n")

#anal\_file.write(input\_text)

anal\_file.write("\n\n")

anal\_file.write("Вариант 17: Быстрая сортировка. Кириллица. По алфавиту. По возрастанию. Игнорировать числа и знаки препинания. \n")

anal\_file.write(f"Количество слов: {len(words)} \n")

anal\_file.write(f"Время сортировки: {sort\_time\_mlsec:.0f} миллисекунд \n")

anal\_file.write("Статистика (кол-во слов на каждую букву алфавита) \n")

# "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдеёжзийклмнопрстуфхцчшщъыьэюя"

for letter in "абвгдеёжзийклмнопрстуфхцчшщъыьэюя":

anal\_file.write(f"{letter} - {words\_per\_letter(letter,words)}\n")

# Выводим массив слов

print(words)

# 

# Результат выполнения программы.

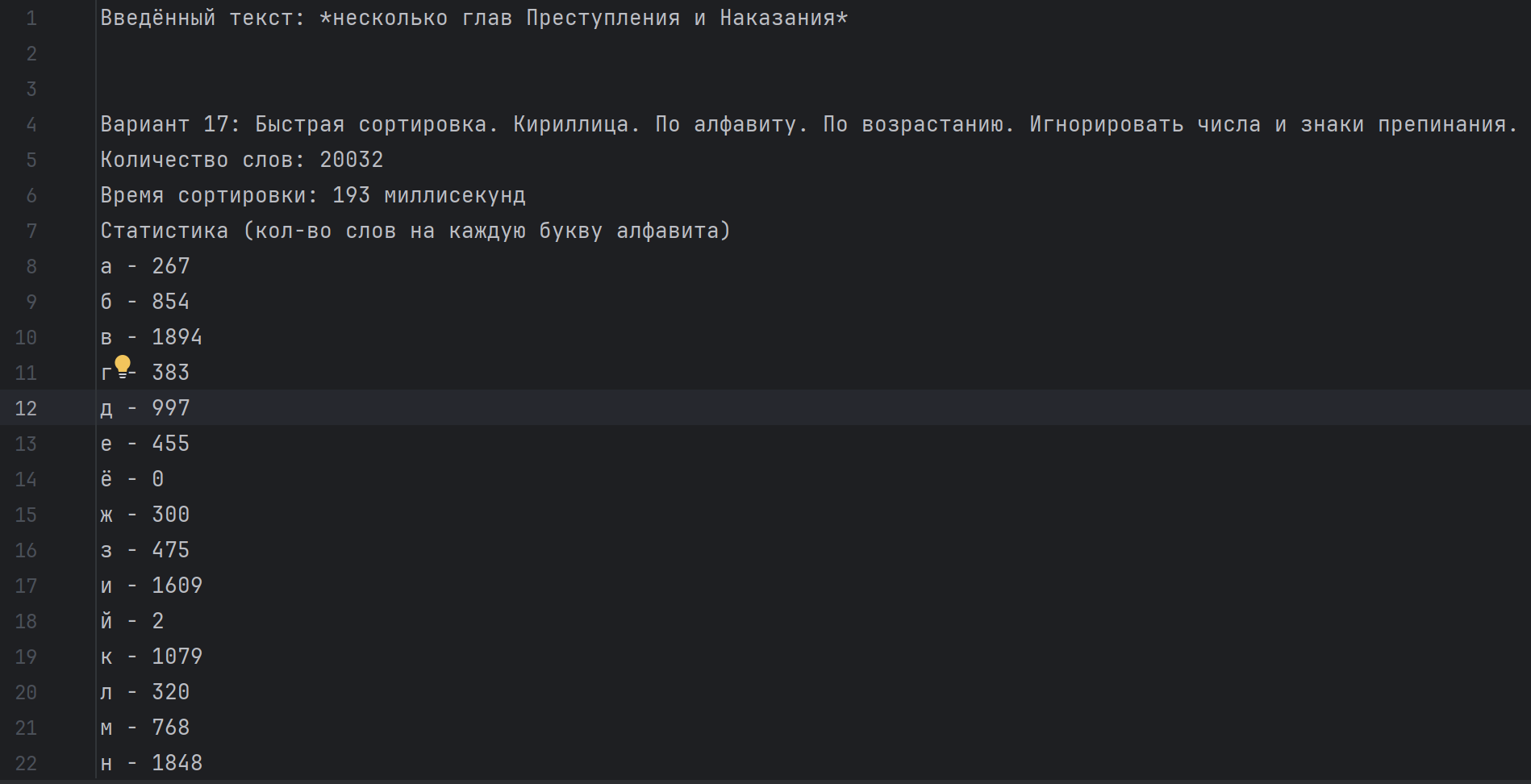


Рисунок 2.1. - 1-ый результат работы программы

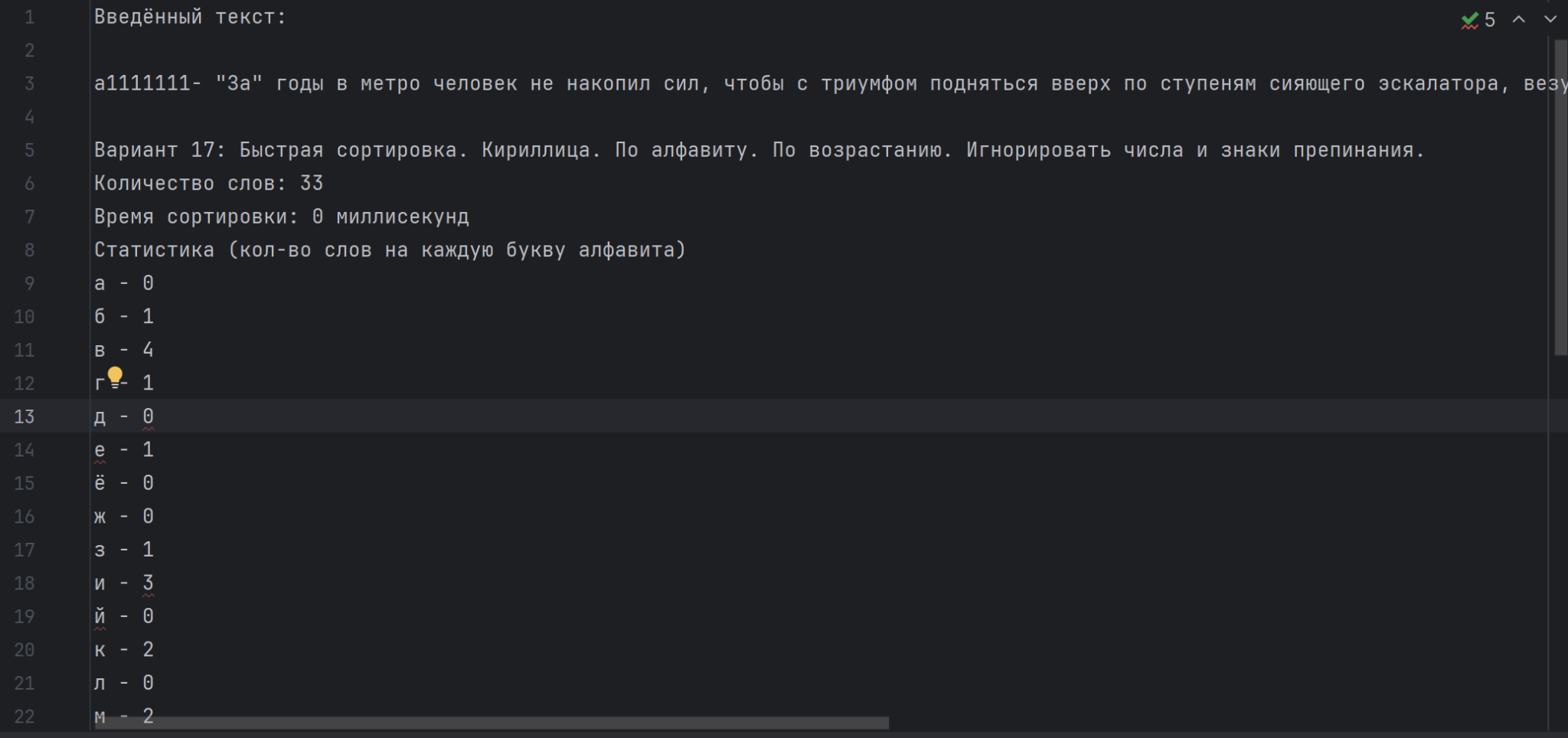


Рисунок 2.2. - 2-ой результат работы программы

# 

# Выводы

В ходе выполнения лабораторной работы мной были освоены и изучены: быстрая сортировка, понятие сложности алгоритма. Написанная программа была протестирована, полученный результат соответствует значению в примере.