Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе №5**

**«Прототип файлового менеджера»**

**Выполнил**:

студент группы 3823Б1ФИ1

Гутянский А.С.

**Проверил**:

доцент кафедры ВВиСТ, к.т.н.,

Сысоев А.В.

Нижний Новгород

2023

Содержание

[Введение 3](#_Toc152597319)

[1. Постановка задачи 4](#_Toc152597320)

[2. Руководство пользователя 5](#_Toc152597321)

[2.1. Стандартный режим использования 5](#_Toc152597322)

[2.2. Режим тестирования производительности 7](#_Toc152597323)

[3.1. Описание структуры программы 9](#_Toc152597324)

[3.2. Описание алгоритмов 9](#_Toc152597325)

[3.2.1. Сортировка пузырьком 9](#_Toc152597326)

[3.2.2. Сортировка выбором 9](#_Toc152597327)

[3.2.3. Сортировка вставкой 10](#_Toc152597328)

[3.2.4. Сортировка Хоара (быстрая сортировка) 10](#_Toc152597329)

[3.2.5. Сортировка Шелла 10](#_Toc152597330)

[3.2.6. Сортировка слиянием 11](#_Toc152597331)

[3.2.7. Сортировка подсчетом 11](#_Toc152597332)

[8. Результаты экспериментов 12](#_Toc152597333)

[Заключение 14](#_Toc152597334)

[Литература 15](#_Toc152597335)

[Приложение 16](#_Toc152597336)

# Введение

Хранение цифровой информации является одной из основных задач современных компьютерных систем. Для хранения информации компьютеры используют различного вида носители. Данные на носителях хранятся в двоичном виде, чаще всего в виде файлов.

Файл – именованная область данных на носителе. В операционных системах файл является базовой единицей хранения информации в файловой системе. Помимо области в памяти и названия операционная система сохраняет для файлов дополнительные данные, такие как время создания, время последнего изменения, права доступа.

Для взаимодействия с файловой системой существуют файловые менеджеры. Они позволяют с легкостью просматривать, искать, сортировать, удалять, перемещать, копировать файлы. Опираясь на ранее сказанное, стоит отметить, что файловый менеджер является одной из важнейших прикладных программ в операционной системе.

Функционал сортировки файлов, предоставляемый файловым менеджером, позволяет проводить полезные действия по администрированию операционной системы, например, поиск файлов большого размера для последующего их удаления в целях освобождения памяти.

# Постановка задачи

Разработать прототип файлового менеджера с функцией показа файлов в заданном каталоге, упорядоченных по возрастанию/убыванию размера.

Входные данные:

* Путь до директории, в которой необходимо отсортировать содержимое.
* Метод сортировки.

Выходные данные:

* Отсортированный список имен файлов с указанием размера.
* Время сортировки.

Программа должна предоставлять пользователю возможность сменить метод сортировки и повторно формировать выходные данные.

Программа должна реализовывать диалог с пользователем посредством интерфейса, который включает:

* возможность ввода пути до заданного каталога;
* возможность выбора метода сортировки;
* возможность просмотра отсортированного списка файлов с указанием размера.

Список методов сортировки:

* пузырьком
* выбором
* вставками
* слиянием
* Хоара
* Шелла
* подсчетом

# Руководство пользователя

## Стандартный режим использования

Перейдите в папку с программой и приложением и запустите файл *task5.exe.* В открывшемся приложении нужно ввести путь к папке, в которой нужно отсортировать файлы.

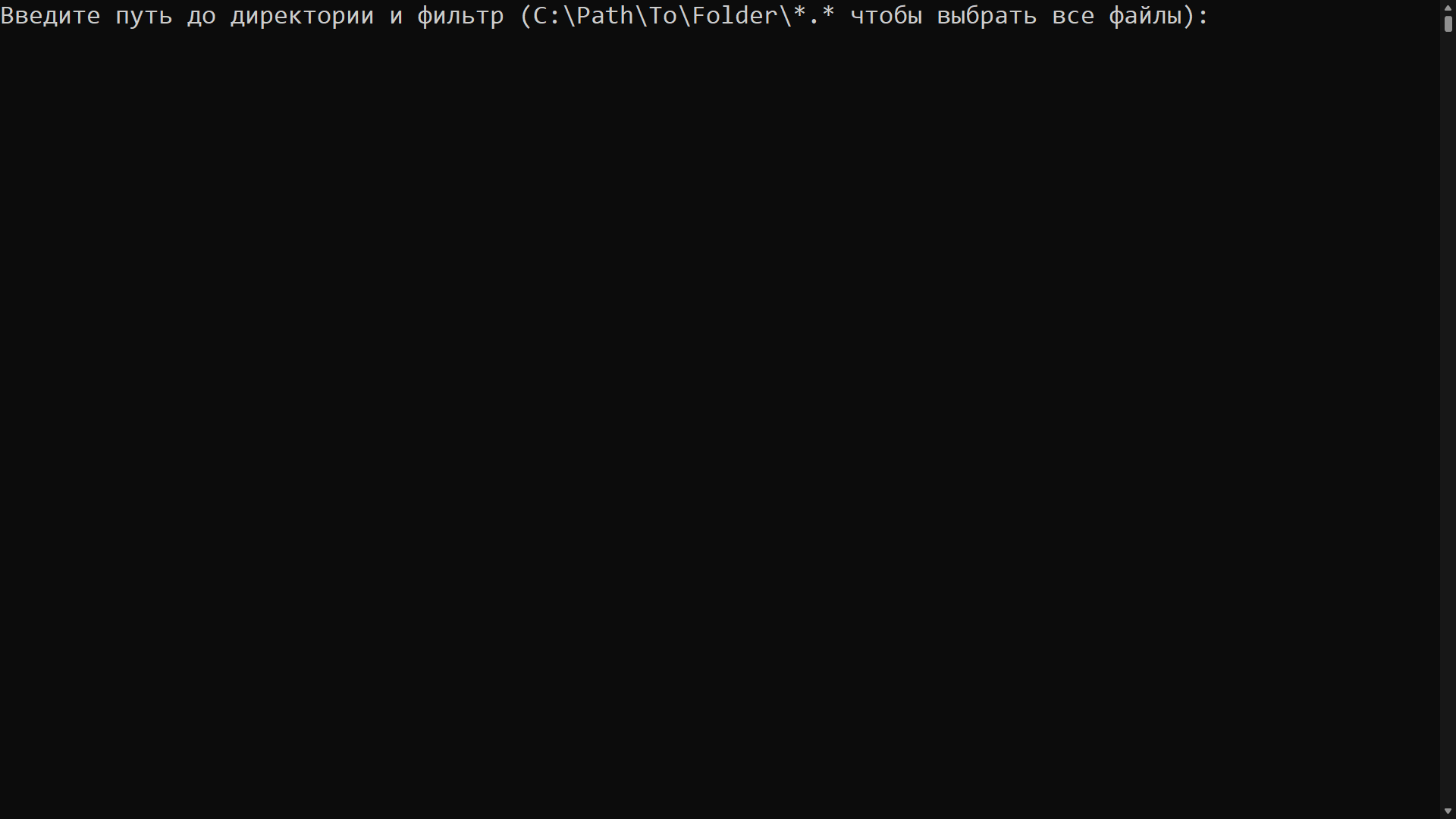


Рис.1 Начальное состояние приложения

Введите полный путь к папке, в которой нужно отсортировать файлы. Путь должен начинаться с буквы диска, где содержится папка, разделитель пути, в качестве разделителя используйте символ «\» (обратный слеш).

В конце пути укажите маску для поиска файлов в папке. Для выбора всех файлов укажите *\*.\*,* а, например, для выбора только изображений PNG-формата укажите *\*.png*.

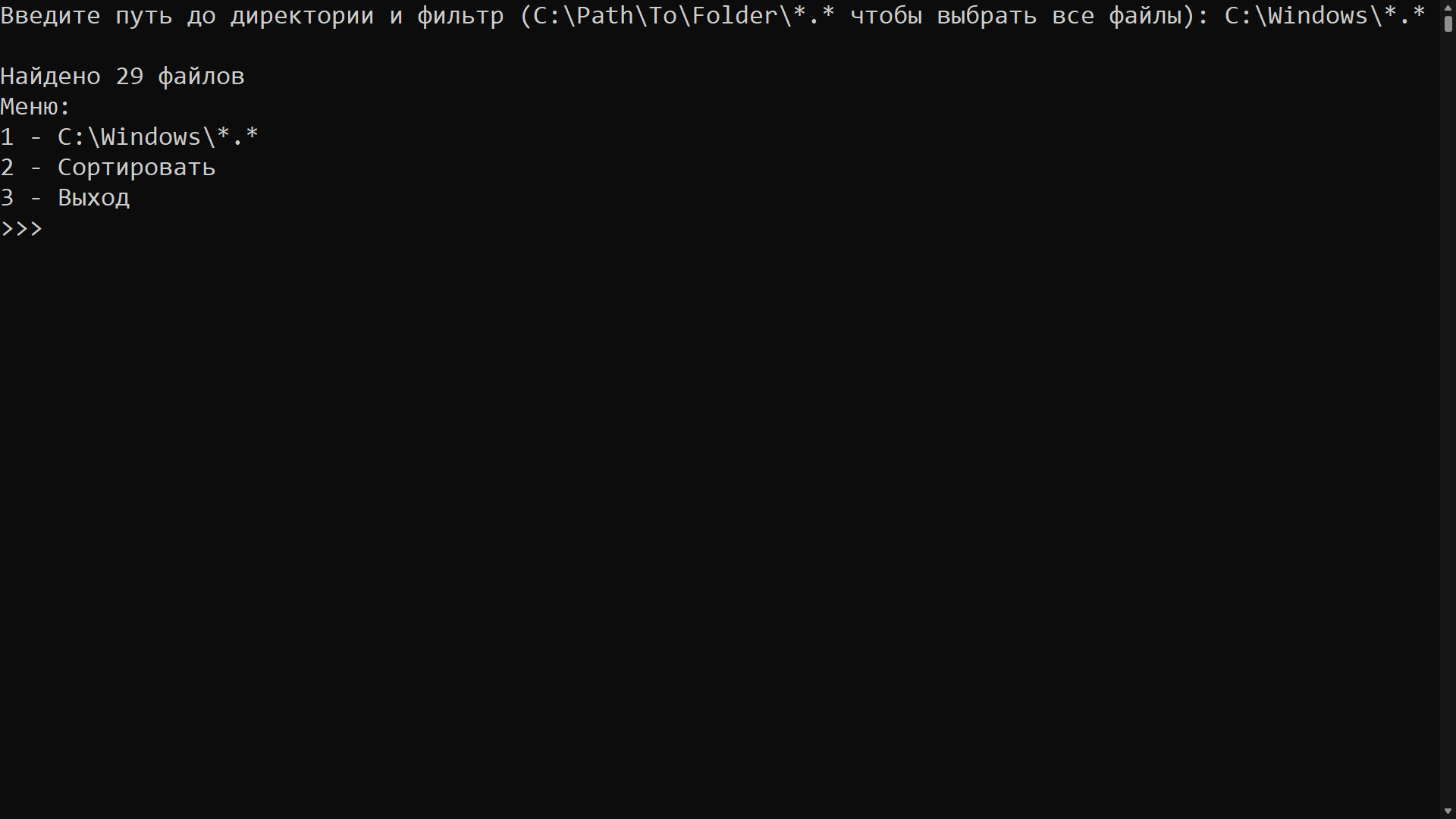


Рис.2 Главное меню приложения

После указания папки будет отображено меню, содержашее 3 пункта, а также количество файлов, найденных в папке. Если приложение не нашло файлов, но вы уверены, что в папке есть файлы, то перепроверьте указанный путь.

Чтобы ввести путь к папке заново, напишите *1* и нажмите *Enter.*

Чтобы выйти из приложения, напишите *3* и нажмите *Enter*.

Для того, чтобы увидеть отсортированный метод файлов, нажмите *2*, затем *Enter.*

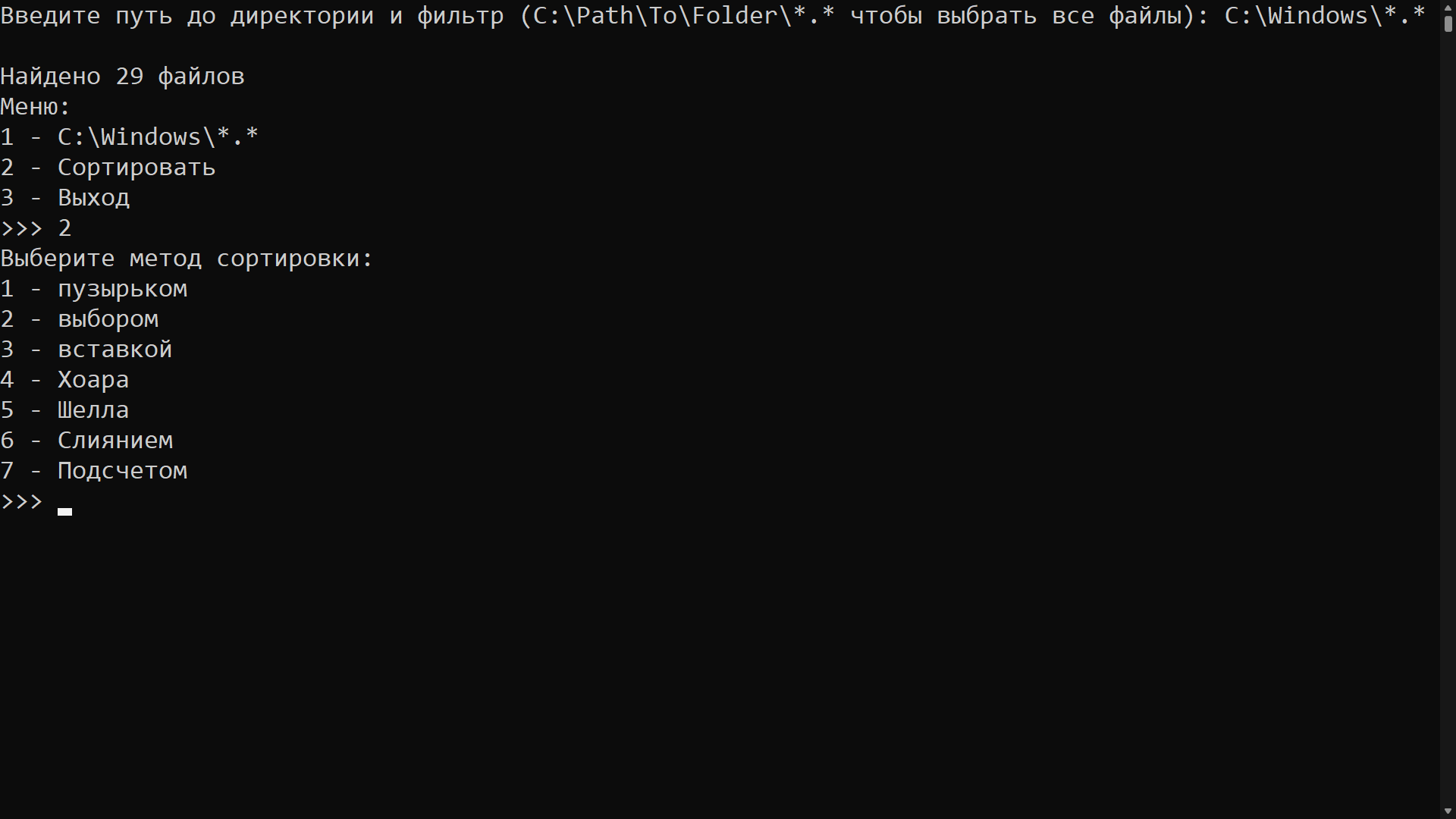


Рис.3 Меню выбора метода сортировки

Выберите один из предлагаемых методов сортировки, нажмите цифру, написанную рядом с названием, затем *Enter*.

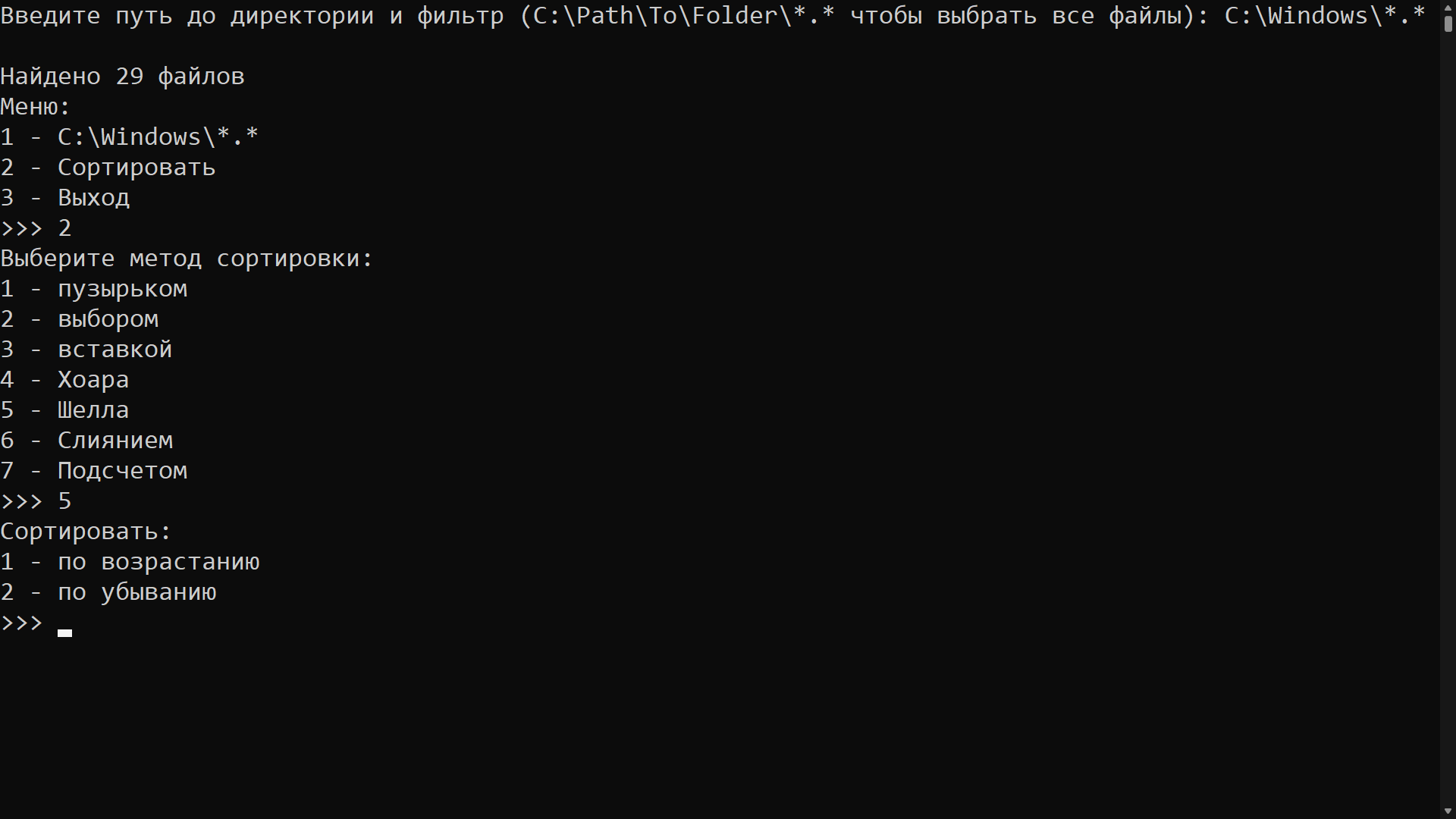


Рис.4 Меню выбора направления сортировки

Выберите направление, сортировки. Чтобы файлы большего размера оказались наверху списка, нажмите *1,* иначе нажмите *2.* Для подтверждения выбора нажмите *Enter.*

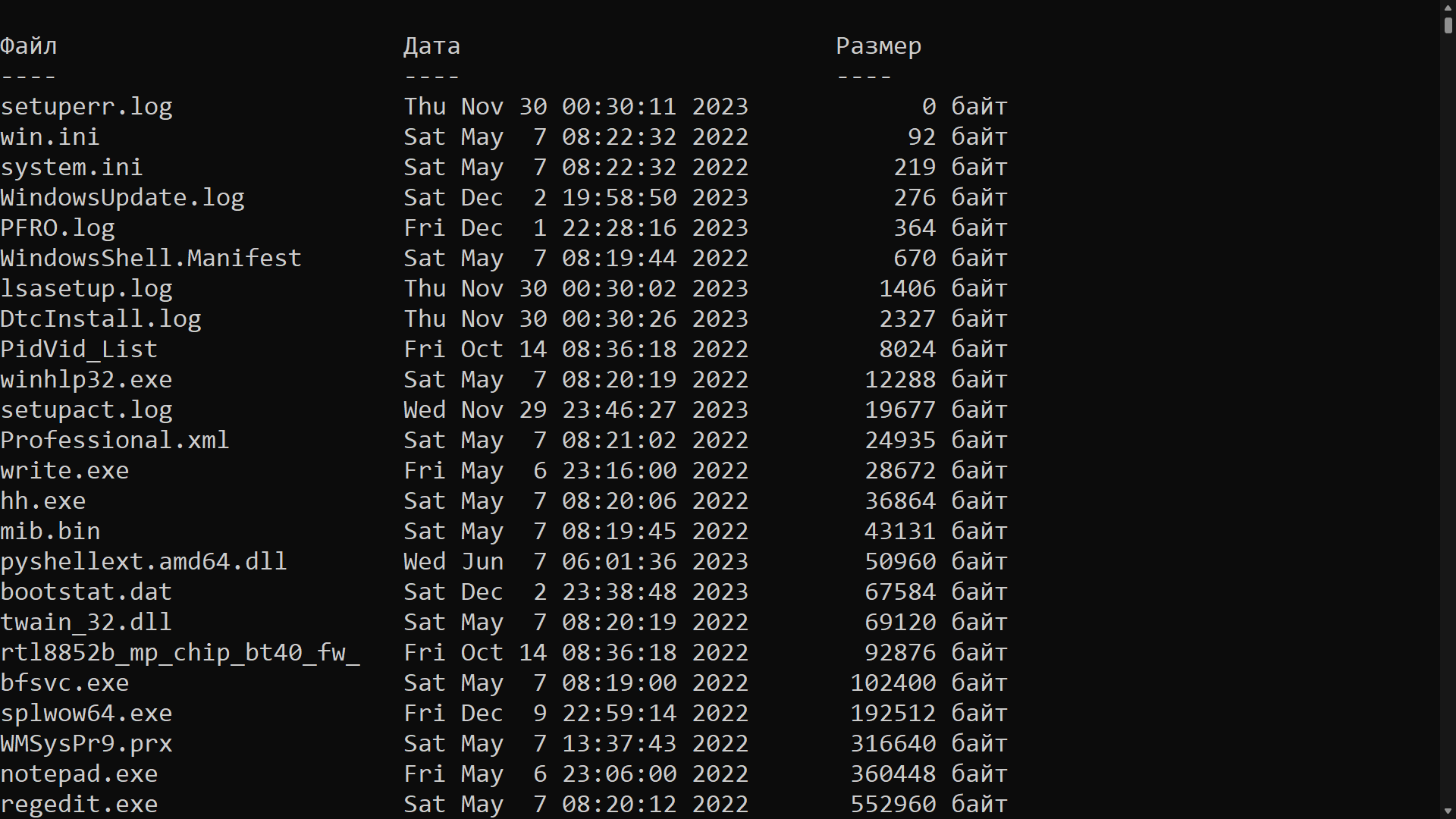


Рис.5 Отсортированный список файлов

После завершения обработки, вы увидите список файлов, найденных в указанной в начале папке и отсортированных по размеру. Для каждого файла программа выводит имя, дату последнего изменения и размер в байтах. После списка программа выводит время, которое заняла сортировка.

Если программа не смогла отсортировать файлы и вывела соответствующее сообщение (это часто случается, когда выбрана сортировка подсчетом и при этом в папке есть файлы большого размера), тогда попробуйте использовать другой метод сортировки, либо выбрать не все файлы из директории, выше описано, как это можно сделать.

## Режим тестирования производительности

Перейдите в папку с программой и приложением и запустите файл *task5.exe “PATH\TO\DIR\\*.\*”*, где в качестве аргумента указывается путь к папке, на которой нужно протестировать производительность приложения (в начале прошлого подраздела есть описание, как правильно указывать путь к папке).

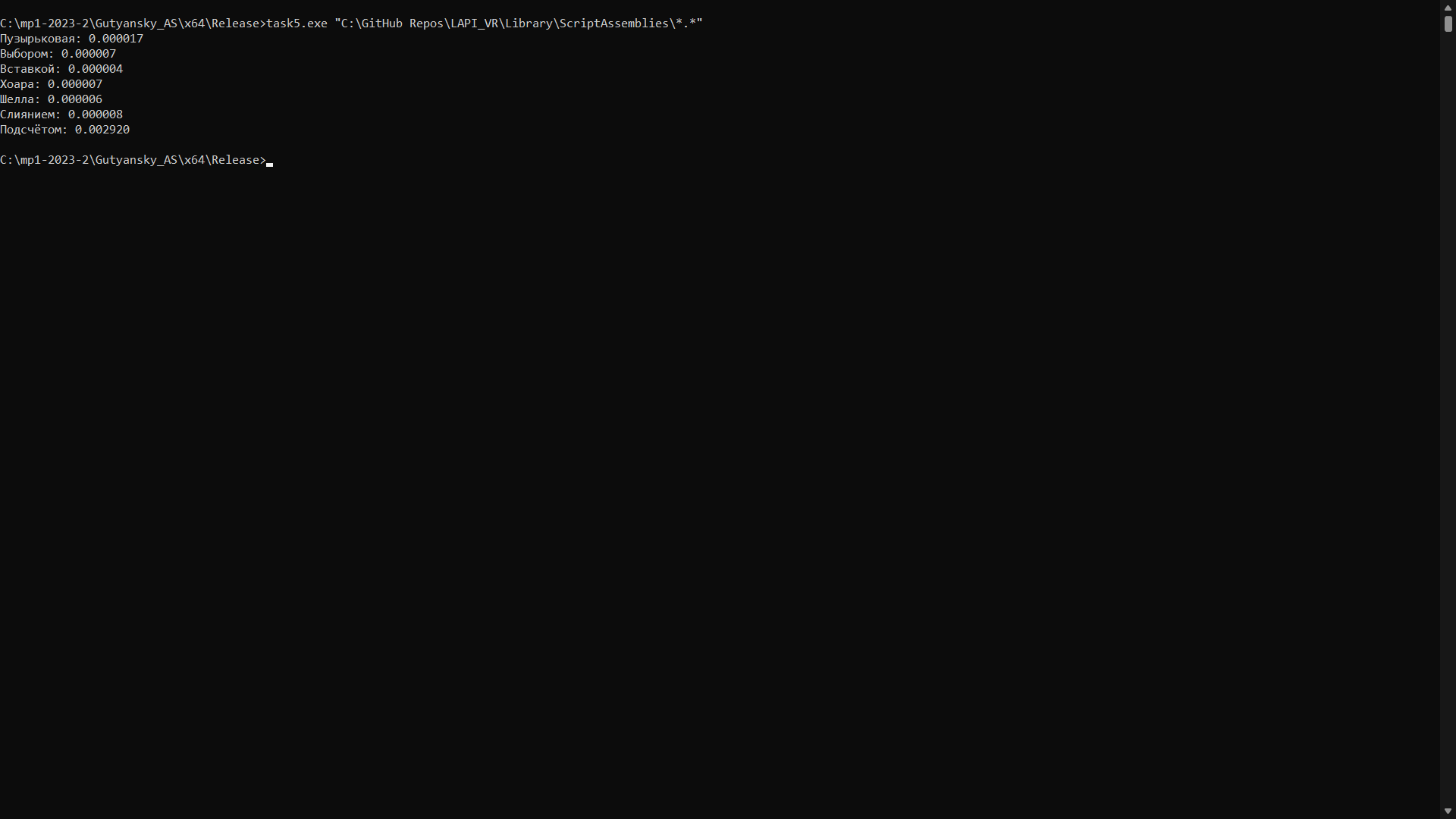


Рис.6 Результаты тестирования производительности

Программа протестирует каждый метод сортировки на файлах, содержащихся по указанному пути, и выдаст затраченное на обработку время в секундах. Если программа вывела *-1* для какого-то метода*,* тогда сортировка файлов этим методом не может быть проведена.

1. **Руководство программиста**

## Описание структуры программы

Программа состоит из трёх модулей. В основном модуле *Source* содержится точка входа *main(),* где происходит выбор режима работы приложения в зависимости от количества переданных параметров. Если передан параметр, тогда запускается режим тестирования производительности, иначе программа через консольный интерфейс запрашивает у пользователя путь к папке, после чего переходит в главное меню.

Основной модуль программы содержит функции *main\_menu()* и *sort\_menu(),* обрабатывающие соответственно главное меню и меню сортировки. Главное меню позволяет выбрать другой путь, вызвать меню сортировки или выйти из приложения. Меню сортировки позволяет выбрать метод сортировки, направление сортировки, после чего выполняет сортировку файлов.

Выбор пунктов меню происходит через ввод пользователем номера пункта. Для этого используется функция *try\_read\_int()*, обрабатывающая ввод от пользователя, при этом попытки ввода продолжаются до тех пор, пока не будет введено корректное число (реализуется в программе через цикл с предусловием).

Сортировка файлов происходит с помощью алгоритмов, описанных в модуле *sorting\_algorithms*. Описание работы данных алгоритмов будет представлено далее далее.

Также в основном модуле после ввода пользователем пути происходит загрузка найденных файлов. Файлы загружаются с помощью обращения к функциям поиска файлов в файловой системе, после чего помещаются в массив структур *FileData,* описанных в одноименном модуле.

## Описание алгоритмов

### Сортировка пузырьком

Алгоритм сортировки пузырьком является простым алгоритмом сортировки, который многократно проходит по списку элементов, сравнивая два соседних элемента и меняя их местами, если они стоят в неправильном порядке. С каждым проходом в конец списка «всплывает» наибольший (наименьший) элемент. Процесс продолжается до тех пор, пока весь список не будет отсортирован.

Шаги алгоритма сортировки пузырьком:

1. Проходим по всем элементам списка.
2. Сравниваем каждую пару соседних элементов.
3. Если элементы стоят в неправильном порядке, меняем их местами.
4. Повторяем шаги 1-3 для каждого элемента списка, пока весь список не будет отсортирован.

Алгоритм сортировки пузырьком имеет временную сложность O(), что делает его неэффективным для больших списков. Однако он легко реализуется и понимается, поэтому используется в учебных целях или на небольших списках.

### Сортировка выбором

Алгоритм сортировки выбором также является простым алгоритмом сортировки. Он проходит по списку элементов и на каждом шаге находит минимальный элемент, после чего меняет его местами с первым элементом списка. Затем процесс повторяется для подсписка, начинающегося со второго элемента, и так далее, пока весь список не будет отсортирован.

Шаги алгоритма сортировки выбором:

1. Находим минимальный элемент в списке.
2. Меняем его местами с первым элементом списка.
3. Повторяем шаги 1-2 для подсписка, начинающегося со второго элемента.
4. Продолжаем этот процесс до тех пор, пока весь список не будет отсортирован.

Алгоритм сортировки выбором также имеет временную сложность O(), но он может быть более эффективным на небольших списках по сравнению с алгоритмом сортировки пузырьком.

### 3.2.3. Сортировка вставкой

Алгоритм сортировки вставкой также является простым алгоритмом сортировки. Он проходит по списку элементов и на каждом шаге берет очередной элемент и вставляет его в отсортированную часть списка, сдвигая большие элементы вправо.

Шаги алгоритма сортировки вставкой:

1. Начинаем со второго элемента списка.
2. Берем очередной элемент и сравниваем его с предыдущими элементами в отсортированной части списка.
3. Если текущий элемент меньше предыдущего, меняем их местами и продолжаем сравнивать с предыдущими элементами, пока не найдем его правильное место.
4. Повторяем этот процесс для всех элементов списка.

Алгоритм сортировки вставкой также имеет временную сложность O(), но он может быть более эффективным на небольших списках по сравнению с алгоритмом сортировки выбором.

### 3.2.4. Сортировка Хоара (быстрая сортировка)

Алгоритм сортировки Хоара, также известный как быстрая сортировка, является одним из наиболее эффективных алгоритмов сортировки. Он основан на принципе "разделяй и властвуй" и использует метод разбиения массива на подмассивы.

Шаги алгоритма быстрой сортировки:

1. Выбираем опорный элемент из массива. Обычно это средний элемент, но может быть выбран любой другой.
2. Разбиваем массив на две части: одна содержит элементы меньше опорного, а другая - элементы больше опорного.
3. Рекурсивно применяем алгоритм к обеим частям массива.
4. Объединяем отсортированные части в один отсортированный массив.

Алгоритм быстрой сортировки имеет временную сложность в среднем O(), что делает его очень эффективным для больших массивов. Однако в худшем случае (если опорный элемент выбран неудачно) его временная сложность может быть O().

### 3.2.5. Сортировка Шелла

Алгоритм сортировки Шелла, также известный как метод уменьшения промежутков, является модификацией алгоритма сортировки вставками. Он был разработан Дональдом Шеллом в 1959 году и представляет собой улучшенную версию сортировки вставками, которая работает более эффективно для больших массивов данных.

Шаги алгоритма сортировки Шелла:

1. Выбираем последовательность промежутков (интервалов), по которым будем сравнивать и переставлять элементы массива. Обычно начальная последовательность выбирается как n/2, где n - длина массива, и затем последовательность уменьшается вдвое на каждом шаге. Такой способ хорошо работает на массивах, чья длина – степень двойки. Однако гораздо лучший вариант предложил Р.Седжвик. Его последовательность имеет вид

При использовании таких приращений среднее количество операций: O(), в худшем случае - порядка O().

1. Для каждого промежутка выполняем сортировку вставками для подмассива, состоящего из элементов, находящихся на расстоянии промежутка друг от друга.
2. Повторяем шаг 2, уменьшая длину промежутка, пока не достигнем промежутка длиной 1.

Алгоритм сортировки Шелла имеет временную сложность в среднем O(), что делает его эффективным для средних и больших массивов данных. Однако его временная сложность зависит от выбора последовательности промежутков и может быть сложно предсказать.

### 3.2.6. Сортировка слиянием

Алгоритм сортировки слиянием является эффективным и стабильным методом сортировки, который основан на принципе "разделяй и властвуй". Он разбивает исходный массив на две равные части, рекурсивно сортирует каждую из них, а затем объединяет отсортированные подмассивы в один, также отсортированный.

Шаги алгоритма сортировки слиянием:

1. Разбиваем исходный массив пополам, пока не достигнем массивов длиной 1.
2. Сливаем (объединяем) отсортированные подмассивы в один, сохраняя отсортированность.

Алгоритм сортировки слиянием имеет временную сложность O(), что делает его эффективным для больших массивов данных. Однако он требует дополнительной памяти для хранения промежуточных результатов слияния.

### 3.2.7. Сортировка подсчетом

Алгоритм сортировки подсчётом — это метод сортировки, основанный на подсчёте количества элементов с определенным значением в исходном массиве. Он особенно эффективен для сортировки целых чисел, когда известен диапазон возможных значений, при этом диапазон небольшой

Шаги алгоритма сортировки подсчётом:

1. Находим минимальное и максимальное значения в исходном массиве.
2. Создаем массив подсчета длиной (max-min+1), где каждый элемент будет хранить количество вхождений определенного значения.
3. Проходим по исходному массиву и увеличиваем соответствующий элемент в массиве подсчета.
4. Выполняем накопление в массиве подсчета, чтобы получить информацию о номерах ячеек массива, куда будут помещены элементы при сортировке.
5. Создаем новый массив, в который помещаем элементы из исходного массива в правильном порядке, используя информацию из массива подсчета.

Алгоритм сортировки подсчётом имеет временную сложность O(n+k), где n - количество элементов в исходном массиве, k - диапазон возможных значений. Он не использует сравнения элементов, что делает его очень эффективным для больших массивов данных с небольшим диапазоном сортируемых значений. Однако он требует дополнительной памяти для массива подсчета.

# Результаты экспериментов

Программа была скомпилирована в *Release* режиме, после чего с помощью скрипта, написанного на *Python*, запущена на разных количествах файлов разных размеров. Отчёт по тестированию программы был сформирован в виде графиков с помощью *matplotlib*. Графики были сглажены усреднением значений соседних по количеству файлов тестов в целях уменьшения шума.

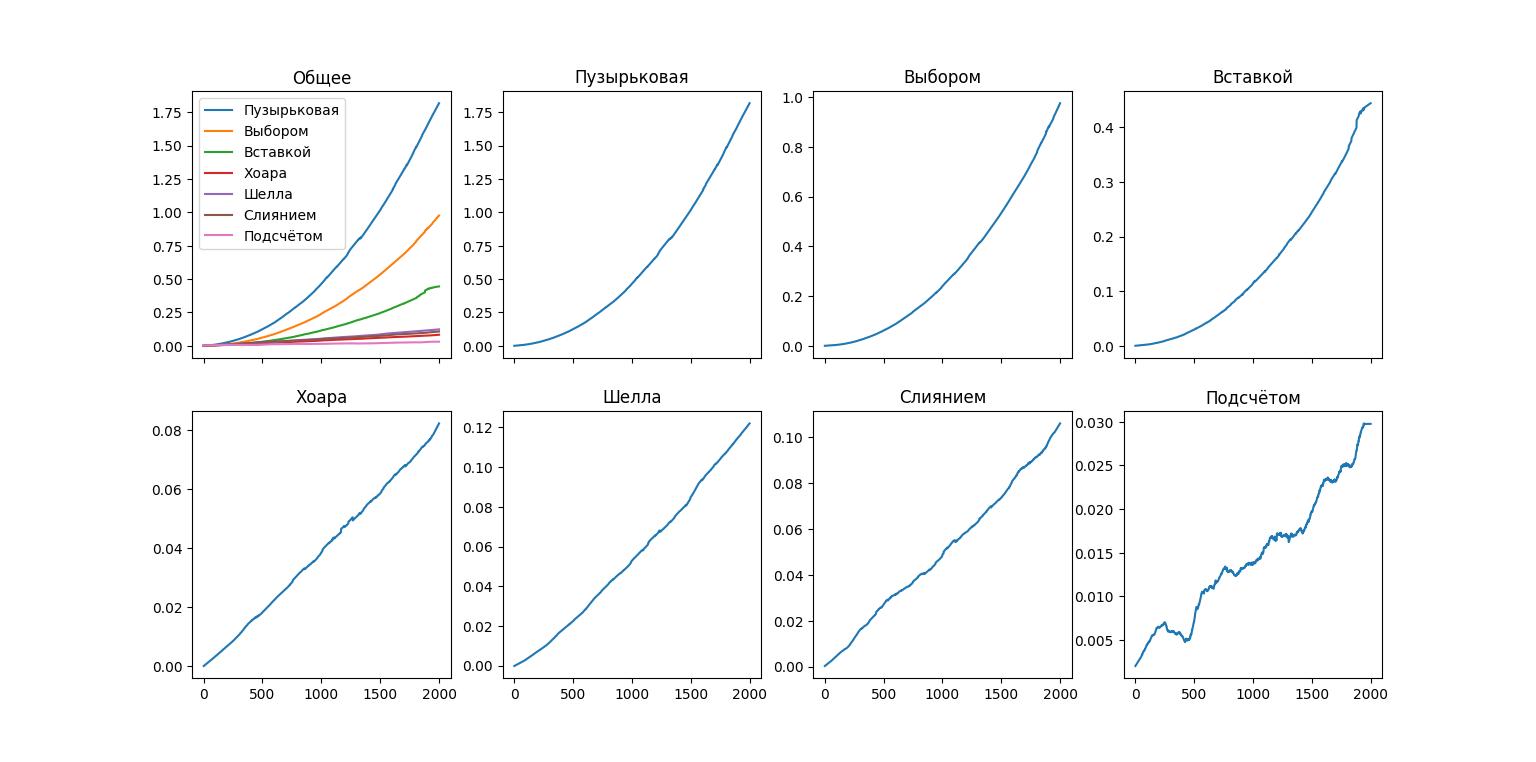


Рис.7 Результаты тестирования методов сортировки

На графиках представлена зависимость времени сортировки от количества файлов. Самой медленной оказалась сортировка пузырьком, а самой быстрой сортировка подсчётом. Время на графиках указано в миллисекундах.

Методы сортировок можно разделить на три группы:

* Квадратичные (пузырьковая, выбором, вставкой). Список из 2000 файлов сортируется примерно за 1,5 мс.
* Логарифмические (Хоара, Шелла, Слиянием). Список из 2000 файлов сортируется примерно за 0,1 мс, что в 15 раз быстрее чем у квадратичных сортирок. График здесь мало отличается от прямой из-за небольшого количества файлов.
* Линейная (подсчётом). Сортирует 2000 файлов за 0.03 мс, но такая производительность достигается только при соблюдении некоторых условий (о них позже). График достаточно шумный из-за того, что максимальный размер файла от теста к тесту меняется случайным образом.

Также были проведены несколько тестов на реальных данных. Они характеризуются небольшим количеством файлов в папке, однако максимальный размер файла достаточно большой (в папке Windows размер достигает 5МБ).

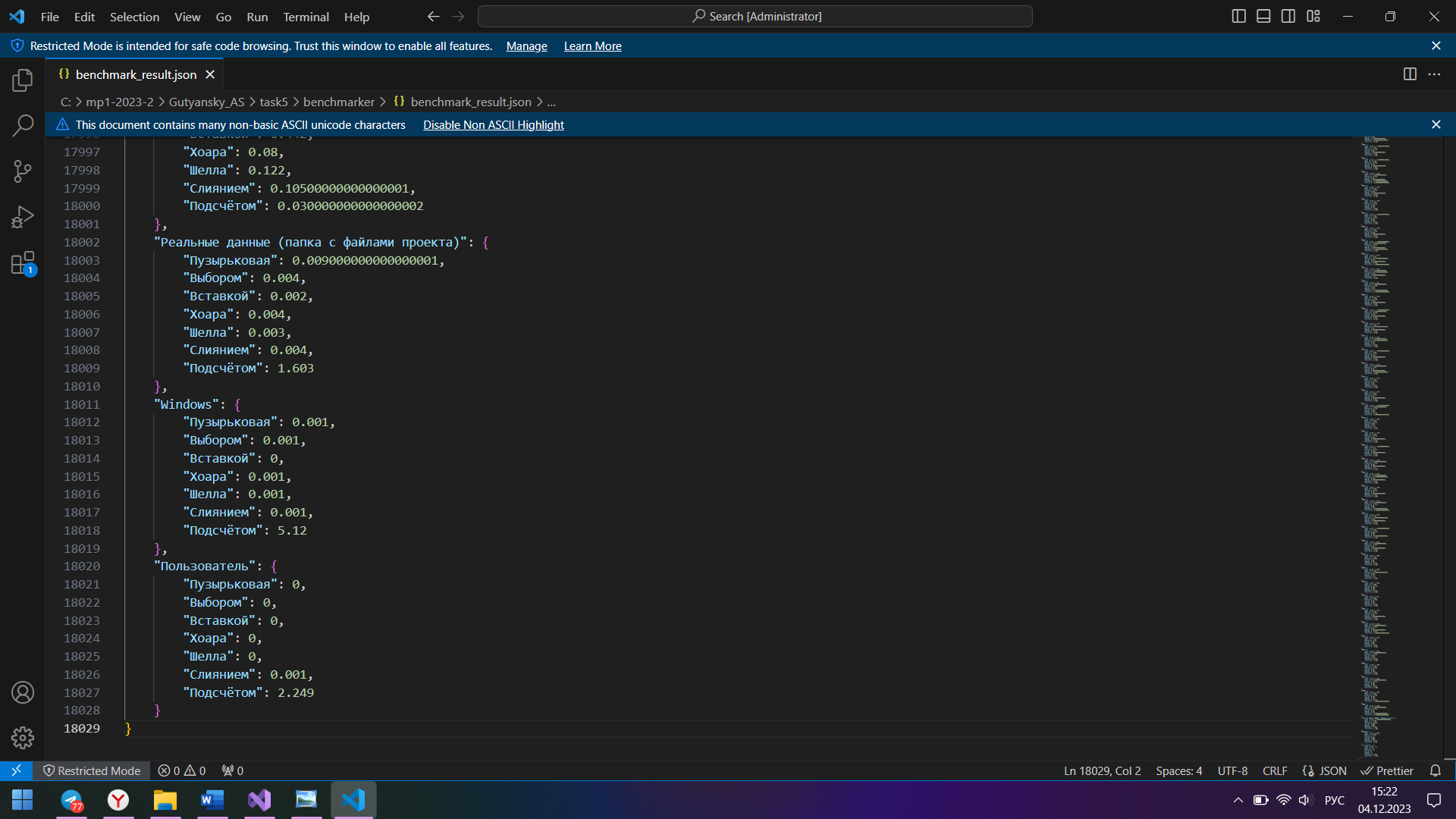


Рис.8 Тесты производительности на реальных данных

Как видим, в отличие от синтетических тестов, в реальных сортировка подсчётом значительно уступает остальным несмотря на то, что является асимптотически более быстрой. Это происходит из-за большого диапазона размеров файлов. Например, если максимальный размер файла в папке 10МБ, тогда алгоритм создаст массив из элементов. В процессе работы несколько раз пройти по нему, что занимает значительное количество времени. Кроме того, размер такого массива занимает место в памяти, что может привести к невозможности выполнения сортировки данным методом, когда максимальный размер файла составляет гигабайты, что является частым явлением в современных реалиях.

# Заключение

В ходе выполнения работы был разработан прототип файлового менеджера. Приложение имеет консольный интерфейс и предоставляет возможность отображения упорядоченного по размеру списка файлов из указываемой пользователем папки. Для упорядочивания списка пользователю предлагается несколько методов сортировки. Наиболее эффективный метод сортировки стоит выбирать в зависимости от наполнения папки. Для папки с огромным количеством файлов небольшого размера (до нескольких МБ) отлично подойдет сортировка подсчетом. В иных случаях стоит применять одну из логарифмических сортировок, например, сортировку Хоара.

# Литература

1. Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). – 2-е изд. – Москва: Вильямс, 2007. – Т. 3. – 832 с.
2. Страуструп Бьерн Язык программирования C++. Специальное издание. Пер. с англ. – М.: Издательство Бином, 2017 г. – 1136 с.
3. Сайт Algolist. Сортировка выбором – <http://algolist.manual.ru/sort/select_sort.php>.
4. Сайт Algolist. Сортировка пузырьком – <http://algolist.manual.ru/sort/bubble_sort.php>.

# Приложение

**Source.c**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <io.h>

#include <string.h>

#include <time.h>

#include <omp.h>

#include <Windows.h>

#include "file\_data.h"

#include "sorting\_algorithms.h"

#define STR\_(X) #X

#define STR(X) STR\_(X)

#define strdup \_strdup

#define MAX\_PATH\_LENGTH 511

size\_t get\_files\_count\_in\_directory(const char\* path);

FileData\* get\_all\_files\_from\_directory(const char\* path, size\_t\* length);

void read\_path(char path[MAX\_PATH\_LENGTH + 1]);

void sort\_menu(char path[MAX\_PATH\_LENGTH + 1]);

void main\_menu(char path[MAX\_PATH\_LENGTH + 1]);

void benchmark(char path[MAX\_PATH\_LENGTH + 1]);

void free\_files(FileData\* files, size\_t length);

void reverse(FileData\* files, int length);

int try\_read\_int(int\* result, int minValue, int maxValue);

int main(int argc, char\* argv[]) {

int arg\_len;

char path[MAX\_PATH\_LENGTH + 1];

if (argc == 2) {

arg\_len = strlen(argv[1]);

if (argv > MAX\_PATH\_LENGTH) {

arg\_len = MAX\_PATH\_LENGTH;

}

strncpy(path, argv[1], arg\_len);

path[MAX\_PATH\_LENGTH] = '\0';

benchmark(path);

return 0;

}

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

read\_path(path);

main\_menu(path);

return 0;

}

void benchmark(char path[MAX\_PATH\_LENGTH + 1]) {

int sort\_success;

size\_t count;

FileData\* files;

double begin, end, elapsed\_time;

// Bubble sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = bubble\_sort(files, count);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Пузырьковая: %lf\n", elapsed\_time);

/// ----------

// Select sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = select\_sort(files, count);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Выбором: %lf\n", elapsed\_time);

/// ----------

// Insert sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = insert\_sort(files, count);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Вставкой: %lf\n", elapsed\_time);

/// ----------

// Quick sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = quick\_sort(files, count);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Хоара: %lf\n", elapsed\_time);

/// ----------

// Shell sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = shell\_sort(files, count);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Шелла: %lf\n", elapsed\_time);

/// ----------

// Merge sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = merge\_sort(files, 0, count, NULL);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Слиянием: %lf\n", elapsed\_time);

/// ----------

// Bubble sort

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

sort\_success = count\_sort(files, count);

end = omp\_get\_wtime();

free\_files(files, count);

elapsed\_time = end - begin;

if (!sort\_success) elapsed\_time = -1;

printf("Подсчётом: %lf\n", elapsed\_time);

/// ----------

}

void read\_path(char path[MAX\_PATH\_LENGTH + 1]) {

int path\_read\_status;

do {

printf("Введите путь до директории и фильтр (С:\\Path\\To\\Folder\\\*.\* чтобы выбрать все файлы): ");

path\_read\_status = scanf("%" STR(MAX\_PATH\_LENGTH) "[^\n]", path);

while (getchar() != '\n');

} while (path\_read\_status != 1);

printf("Найдено %llu файлов\n", get\_files\_count\_in\_directory(path));

}

void main\_menu(char path[MAX\_PATH\_LENGTH + 1]) {

int user\_choice;

while (1) {

do {

printf("Меню:\n");

printf("1 - %s\n", path);

printf("2 - Сортировать\n");

printf("3 - Выход\n");

printf(">>> ");

} while (!try\_read\_int(&user\_choice, 1, 3));

switch(user\_choice) {

case 1:

read\_path(path);

break;

case 2:

sort\_menu(path);

break;

case 3:

return;

}

}

}

void sort\_menu(char path[MAX\_PATH\_LENGTH + 1]) {

int sort\_success;

int sort\_method;

int reverse\_sort;

size\_t count;

FileData\* files;

double begin, end, elapsed\_time;

char time\_buffer[30];

do {

printf("Выберите метод сортировки:\n");

printf("1 - пузырьком\n");

printf("2 - выбором\n");

printf("3 - вставкой\n");

printf("4 - Хоара\n");

printf("5 - Шелла\n");

printf("6 - Слиянием\n");

printf("7 - Подсчетом\n");

printf(">>> ");

} while (!try\_read\_int(&sort\_method, 1, 7));

do {

printf("Сортировать:\n");

printf("1 - по возрастанию\n");

printf("2 - по убыванию\n");

printf(">>> ");

} while (!try\_read\_int(&reverse\_sort, 1, 2));

files = get\_all\_files\_from\_directory(path, &count);

begin = omp\_get\_wtime();

switch (sort\_method) {

case 1:

sort\_success = bubble\_sort(files, count);

break;

case 2:

sort\_success = select\_sort(files, count);

break;

case 3:

sort\_success = insert\_sort(files, count);

break;

case 4:

sort\_success = quick\_sort(files, count);

break;

case 5:

sort\_success = shell\_sort(files, count);

break;

case 6:

sort\_success = merge\_sort(files, 0, count, NULL);

break;

case 7:

sort\_success = count\_sort(files, count);

break;

}

if (reverse\_sort == 2) {

reverse(files, count);

}

end = omp\_get\_wtime();

elapsed\_time = end - begin;

if (!sort\_success) {

printf("Не удалось отсортировать файлы!\n");

}

else {

printf("Файлы\n\n");

printf("Файл %20c Дата %22c Размер\n", ' ', ' ');

printf("---- %20c ---- %22c ----\n", ' ', ' ');

for (int i = 0; i < count; i++) {

ctime\_s(time\_buffer, \_countof(time\_buffer), &(files[i].time\_write));

printf("%-25.25s %-24.24s %10lu байт\n", files[i].name, time\_buffer, files[i].size);

}

if (count == 0) {

printf("Файлы не найдены! Попробуйте выбрать другую директорию.\n");

}

printf("Время сортировки: %lfsec\n", elapsed\_time);

}

free\_files(files, count);

}

size\_t get\_files\_count\_in\_directory(const char\* path) {

struct \_finddata\_t c\_file;

intptr\_t hFile;

size\_t count = 0;

if ((hFile = \_findfirst(path, &c\_file)) == -1L)

return 0;

else

{

do {

if (!(c\_file.attrib & \_A\_SUBDIR)) {

count++;

}

} while (\_findnext(hFile, &c\_file) == 0);

\_findclose(hFile);

}

return count;

}

FileData\* get\_all\_files\_from\_directory(const char\* path, size\_t\* length) {

struct \_finddata\_t c\_file;

intptr\_t hFile;

FileData\* files;

int index;

\*length = get\_files\_count\_in\_directory(path);

index = 0;

if (\*length == 0 || (hFile = \_findfirst(path, &c\_file)) == -1L) {

return NULL;

}

files = (FileData\*)malloc(sizeof(FileData) \* \*length);

do {

if (!(c\_file.attrib & \_A\_SUBDIR)) {

if (index >= \*length) {

break;

}

FileData data;

data.name = strdup(c\_file.name);

data.time\_write = c\_file.time\_write;

data.size = c\_file.size;

files[index++] = data;

}

} while (\_findnext(hFile, &c\_file) == 0);

\_findclose(hFile);

return files;

}

void free\_files(FileData\* files, size\_t length) {

size\_t i;

for (i = 0; i < length; i++) {

free(files[i].name);

}

free(files);

}

void reverse(FileData\* files, int length) {

int i;

for (i = 0; i < length / 2; i++) {

swap(files, i, length - i - 1);

}

}

int try\_read\_int(int\* result, int minValue, int maxValue) {

int read\_arguments;

read\_arguments = scanf\_s("%d", result);

while (getchar() != '\n');

if (read\_arguments == 1 && \*result >= minValue && \*result <= maxValue) {

return 1;

}

return 0;

}

**file\_data.h**

#pragma once

#include <time.h>

#include <io.h>

typedef struct {

char\* name;

time\_t time\_write;

\_fsize\_t size;

} FileData;

**sorting\_algorithms.h**

#pragma once

#include "file\_data.h"

void swap(FileData\* files, int i, int j);

int bubble\_sort(FileData\* files, int length);

int select\_sort(FileData\* files, int length);

int insert\_sort(FileData\* files, int length);

int quick\_sort(FileData\* files, int length);

int shell\_sort(FileData\* files, int length);

int merge\_sort(FileData\* files, int lower\_bound, int upper\_bound, FileData\* buffer);

int count\_sort(FileData\* files, int length);

**sorting\_algorithms.c**

#include "sorting\_algorithms.h"

#include <stdlib.h>

void swap(FileData\* files, int i, int j) {

FileData temp;

temp = files[i];

files[i] = files[j];

files[j] = temp;

}

int bubble\_sort(FileData\* files, int length) {

int i, j, swaps;

for (i = 0; i < length - 1; i++) {

swaps = 0;

for (j = 0; j < length - i - 1; j++) {

if (files[j].size > files[j + 1].size) {

swap(files, j, j + 1);

swaps++;

}

}

if (swaps == 0) {

break;

}

}

return 1;

}

int select\_sort(FileData\* files, int length) {

int i, j, min\_index;

\_fsize\_t min\_size;

for (i = 0; i < length; i++) {

min\_index = i;

min\_size = files[i].size;

for (j = i + 1; j < length; j++) {

if (files[j].size < min\_size) {

min\_index = j;

min\_size = files[j].size;

}

}

swap(files, i, min\_index);

}

return 1;

}

int insert\_sort(FileData\* files, int length) {

int i, j;

FileData current\_element;

for (i = 1; i < length; i++) {

current\_element = files[i];

j = i - 1;

while (j >= 0 && files[j].size > current\_element.size) {

files[j + 1] = files[j];

--j;

}

files[j + 1] = current\_element;

}

return 1;

}

int quick\_sort(FileData\* files, int length) {

int i, j;

\_fsize\_t center\_size;

if (length == 0) return 1;

i = 0;

j = length - 1;

center\_size = files[length / 2].size;

do {

while (files[i].size < center\_size) i++;

while (files[j].size > center\_size) j--;

if (i <= j) {

swap(files, i, j);

i++;

j--;

}

} while (i <= j);

if (j > 0) quick\_sort(files, i);

if (length > i) quick\_sort(files + i, length - i);

return 1;

}

int increment(int inc[], int length) {

int p1, p2, p3, s;

p1 = p2 = p3 = 1;

s = -1;

do {

if (++s % 2) {

inc[s] = 8 \* p1 - 6 \* p2 + 1;

}

else {

inc[s] = 9 \* p1 - 9 \* p3 + 1;

p2 \*= 2;

p3 \*= 2;

}

p1 \*= 2;

} while (3 \* inc[s] < length);

return s > 0 ? --s : 0;

}

int shell\_sort(FileData\* files, int length) {

FileData temp;

int inc, s, i, j, seq[40];

if (length == 0) return 1;

s = increment(seq, length);

while (s >= 0) {

inc = seq[s--];

for (i = inc; i < length; i++) {

temp = files[i];

for (j = i - inc; (j >= 0) && (files[j].size > temp.size); j -= inc)

files[j + inc] = files[j];

files[j + inc] = temp;

}

}

return 1;

}

int merge(FileData\* files, int lower\_bound, int split\_index, int upper\_bound, FileData\* buffer) {

int left\_index, right\_index, merge\_index;

left\_index = lower\_bound;

right\_index = split\_index;

merge\_index = 0;

while (left\_index < split\_index && right\_index < upper\_bound) {

if (files[left\_index].size < files[right\_index].size)

buffer[merge\_index++] = files[left\_index++];

else

buffer[merge\_index++] = files[right\_index++];

}

while (right\_index < upper\_bound)

buffer[merge\_index++] = files[right\_index++];

while (left\_index < split\_index)

buffer[merge\_index++] = files[left\_index++];

for (merge\_index = 0; merge\_index < upper\_bound - lower\_bound; merge\_index++)

files[lower\_bound + merge\_index] = buffer[merge\_index];

return 1;

}

int merge\_sort(FileData\* files, int lower\_bound, int upper\_bound, FileData\* buffer) {

int success;

int split;

int buffer\_created;

if (upper\_bound - lower\_bound > 1) {

buffer\_created = 0;

if (buffer == NULL) {

buffer = malloc(sizeof(FileData) \* (upper\_bound - lower\_bound));

if (buffer == NULL) {

return 0;

}

buffer\_created = 1;

}

split = (lower\_bound + upper\_bound) / 2;

merge\_sort(files, lower\_bound, split, buffer);

merge\_sort(files, split, upper\_bound, buffer);

success = merge(files, lower\_bound, split, upper\_bound, buffer);

if (!success) {

if (buffer\_created) {

free(buffer);

buffer = NULL;

}

return 0;

}

if (buffer\_created) {

free(buffer);

buffer = NULL;

}

}

return 1;

}

int count\_sort(FileData\* files, int length) {

int i;

\_fsize\_t j;

\_fsize\_t max\_size = 0;

int\* counts;

FileData\* temp;

if (length == 0) return 1;

for (i = 0; i < length; i++) {

if (files[i].size > max\_size) {

max\_size = files[i].size;

}

}

counts = (int\*)calloc(max\_size + 1, sizeof(int));

if (counts == NULL) {

return 0;

}

for (i = 0; i < length; i++) {

counts[files[i].size]++;

}

for (j = 1; j <= max\_size; j++) {

counts[j] += counts[j - 1];

}

temp = (FileData\*)malloc(sizeof(FileData) \* length);

if (temp == NULL) {

free(counts);

return 0;

}

for (i = 0; i < length; i++) {

temp[counts[files[i].size] - 1] = files[i];

counts[files[i].size]--;

}

for (i = 0; i < length; i++) {

files[i] = temp[i];

}

free(counts);

free(temp);

return 1;

}